Improving Reuse of Component Families by Generating Component Hierarchies

Marko Rosenmüller
School of Computer Science,
University of Magdeburg,
Germany
rosenmue@ovgu.de

Norbert Siegmund
School of Computer Science,
University of Magdeburg,
Germany
nsiegmun@ovgu.de

Martin Kuhlemann
School of Computer Science,
University of Magdeburg,
Germany
mkuhlema@ovgu.de

ABSTRACT

Feature-oriented software development (FOSD) enables developers to generate families of similar components. However, current FOSD approaches degrade component reuse because they do not allow a developer to combine multiple components of the same family in a larger program. This is because individual family members cannot be distinguished from each other. We present an approach to model and generate *component hierarchies* that allow a programmer to combine multiple component variants. A component hierarchy structures the components of a family according to their functionality. Due to subtyping between the components of a hierarchy, client developers can write generic code that works with different component variants.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.13 [**Software Engineering**]: Reusable Software—*Domain engineering, Reusable libraries*

General Terms

Design, Languages

1. INTRODUCTION

Scalable software reuse can be achieved by developing components, libraries, and frameworks (in the following referred to as components) as *software product lines (SPLs)* [4]. From a component SPL, programmers can derive a family of similar components that can be distinguished in terms of *features* [6]. Features represent characteristics of a component that are of interest to some stakeholder. A developer can build more complex SPLs by combining multiple component SPLs. This results in a set of interdependent SPLs which we call a *multi product line (MPL)* [21].

In previous work, we presented an approach to model MPLs and to automate their configuration [21]. In this paper, we extend the modeling approach and address the implementation of MPLs with *feature-oriented programming (FOP)* [19, 3]. FOP and other approaches for SPL development do not allow a programmer to use different variants of a component in the same program. The reason is that we cannot distinguish two variants of a component from each other since they are derived from the same code base and use the same names (e.g., for classes). For example, in the customizable DBMS Berkeley DB[1], programmers use the C preprocessor to implement variability. Since a function has the same name in all Berkeley DB variants, we cannot use different variants of the DBMS in the same program (e.g., one for stream processing and one for persistent storage).

As another example consider an SPL for container data structures. We can implement it with FOP to derive different kinds of data structures (e.g., a sorted list and a synchronized list). A client developer should be able to use different variants of the SPL in the same program. Ideally, the data structures should span a type hierarchy to simplify development of generic client code. For example, a sorted list and a synchronized list should be subtypes of a basic list to use them polymorphically. OOP concepts, such as inheritance (e.g., creating a subclass for each kind of list), cannot solve the problem because a programmer has to create subclasses for every class of each component variant [4]. Finally, most FOP approaches completely replicate the code when generating different variants of a component. We summarize the observed problems as follows:

- **Naming Conflicts.** When deriving different variants of a component from an SPL, the names of classes in different variants are the same. Hence, a programmer (and a compiler) cannot distinguish the different variants (e.g., to create objects).
- **Missing Subtyping.** Even though different SPL instances provide a similar interface there is no subtyping relationship between them. This hampers development of generic code in client applications.
- **Code Replication.** Different variants of the same SPL share functionality but there is no code reuse: features that are used in two variants are completely replicated.

We observe these problems when components are embedded in a program (e.g., as statically linked libraries) and used via an API. Hence, we focus on components that are represented in the programming language (e.g., as a set of classes). We define three requirements to enable reuse of different variants of a component within the same program:

[1] http://www.oracle.com/database/berkeley-db/

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FOSD'10 October 10, 2010, Eindhoven, The Netherlands
Copyright 2010 ACM 978-1-4503-0208-1/10/10 ...$10.00.

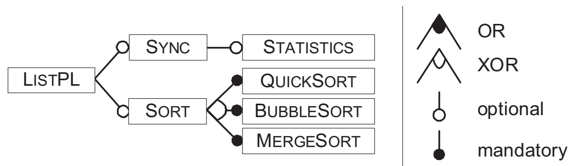

Figure 1: Feature diagram of an SPL for list data structures.

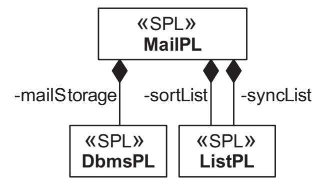

Figure 2: Modeling product lines with UML: A mail client SPL (MailPL) using a DBMS and List SPL.

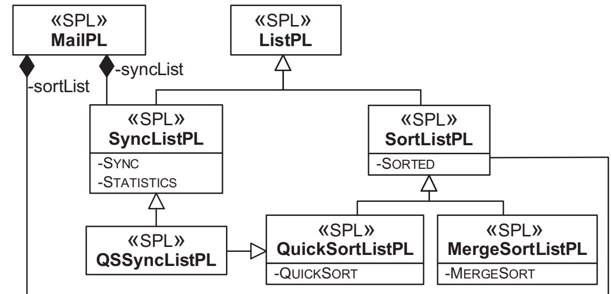

Figure 3: Specialization hierarchy of SPL ListPL. MailPL uses two specializations of ListPL.

1. Instance identification: programmers must be able to distinguish two instances of the same SPL.
2. Subtyping: SPL instances should span a type hierarchy to be used polymorphically.
3. Code Reuse: code should be reused between instances of the same SPL.

Requirement (1) is mandatory for using different component variants in the same program. Requirements (2) and (3) are optional: Requirement (2) simplifies client development because it allows a programmer to write generic code; Requirement (3) is an optimization.

To address the problems already at a conceptual, implementation-independent level, we extend our approach for modeling MPLs. We use *SPL specialization* [7] to define *component hierarchies* and define a subtype relationship between component variants. We then demonstrate how component hierarchies can be generated from FOP code.

2. COMPONENT SPLS

A component SPL allows a programmer to derive a component tailored to her needs. As a running example, consider an SPL for list data structures (ListPL). We can use it to derive different kinds of linked lists such as sorted and synchronized lists. In Figure 1, we show the ListPL *feature diagram* [11, 6], a hierarchical representation of the features. A feature diagram includes relations between features (such as an XOR relation between alternative features) to avoid invalid feature combinations in a concrete SPL instance. For example, feature Sort in Figure 1 represents the functionality for sorting list elements. It has three alternative subfeatures that implement different sort algorithms.

A programmer can use the ListPL to implement a mail client SPL (MailPL). The mail client may also use other SPLs such as a DBMS for mail storage. We call the whole set of interdependent SPLs a *multi product line (MPL)* [21]. The product of an MPL is a set of interacting products of the underlying SPLs. With an *MPL composition model* we describe which SPL instances are used within an MPL. An example for the mail client is shown in Figure 2: MailPL uses an instance of DbmsPL and two different instances of ListPL, which we describe with a composition relationship. SPL instance names (e.g., `sortList`) are used to identify different instances of the same SPL on the model level.

2.1 SPL Specialization Hierarchies

In order to specify which functionality an SPL instance (e.g., `sortList` in Fig. 2) must provide, we use SPL specializations. A specialization of an SPL is a configuration step that eliminates configuration choices [7]. Usually, a specialization does not specify an SPL instance completely; it is only a partial configuration that still provides some variability. In Figure 3, we show an extended ListPL model. We use inheritance to denote specializations of `ListPL`, which results in an *specialization hierarchy.* For example, `SortListPL` and `SyncListPL` are specializations of `ListPL`, each representing a subset of the variants. Feature Sort is included in all instances of `SortListPL`. Hence, we can only derive sorted lists from it. `QuickSortListPL` and `MergeSortListPL` are specializations of `SortListPL` that implement different sorting algorithms. A specialization step does not necessarily add features to an SPL. For example, it may explicitly exclude a feature. In general, arbitrary constraints can be used to create a specialized SPL by reducing the number of valid configurations.

A *fully specialized* SPL represents only a single configuration [7]. We can use it to directly derive the corresponding SPL instance. In contrast, when creating an instance from an incompletely specialized SPL, we have to bind remaining variability first. For example, we can create an instance from `SortListPL` by selecting feature QuickSort and excluding feature Sync.

2.2 Subtyping and SPL Interfaces

The specialization hierarchy defines a subtype relationship between SPLs: A specialized SPL D is a subtype of a less specialized SPL B. If an SPL is a specialization of another SPL, and thus a subtype, can be checked with a SAT solver [23]. Subtyping between SPLs allows us to use them polymorphically. For example, MailPL in Figure 3 uses `SortListPL`, but also accepts every subtype thereof such as an instance of `QuickSortListPL`. Hence, an SPL instance can be used at places where its super type is required.

To describe the interaction between SPLs, we introduce the notion of an *SPL interface.* We distinguish an SPL's *semantic interface* from its *programming interface* (which we introduce in Section 3.3). We define the semantic interface of a (specialized) SPL as the set of features that are present

in all valid instances of the SPL (i.e., the minimally required features). These are mandatory features, features selected via specialization, and features required due to constraints. By adding features in specialization steps we extend the interface of an SPL. However, not every specialization step extends the interface. For example, the interface does usually not change when we add a constraint that excludes a feature. Hence, when the semantic interface of an SPL D (i.e., the set of features) is a superset of the the interface of an SPL B then D is a subytpe of B. On the other hand, when D is a subtype of B then the interface of D is a superset of the interface of B, but not necessarily a *proper* superset.

The expected and required semantic interfaces (i.e., the set of expected and required features) can be used to check whether one component provides the functionality required by another component. For example, we can check whether an instance of ListPL provides all features an instance of MailPL expects. This is a kind of semantic compatibility which is in contrast to the syntactic compatibility that is checked with programming interfaces.

2.3 Summary

Composition models and specialization hierarchies provide means to model MPLs and to distinguish different variants of an SPL at a conceptual level. The requirement to distinguish different instances of an SPL (Req. 1 in Sec. 1) is satisfied by using named SPL instances. The subtype relationship between SPL specializations allows us to use different instances polymorhically (Req. 2). Requirement 3 (Code Reuse) does not apply to the model level. Nevertheless, we can reuse SPL instances (i.e., an SPL's configuration) at different places in an MPL model and thus avoid redefinitions. In the next Section, we show how these concepts can be mapped to the implementation of an SPL.

3. GENERATING COMPONENTS

We demonstrate how the concepts can be applied to SPL implementation for the programming languages *Jak*[2] and *FeatureC++*[3], which are FOP extensions for Java and C++.

3.1 Feature-oriented Programming

FOP allows a programmer to implement the features of an SPL as separate *feature modules*. [19, 3]. Feature modules decompose a class into a base class and *class refinements*. In Figure 4, we depict the FeatureC++ code of the base implementation of a class `List` (Lines 1–6) of the ListPL, and two refinements of the class (Lines 7–16). Elements are added to the list via method `add`. The refinements in features SORT and SYNC extend the base implementation. They override method `add` to implement sorted insertion (Line 8) and synchronization (Line 12). Feature SYNC also adds a new field `sync` to synchronize access to the list. Overridden methods are called with `super` as shown for feature SYNC (Line 14).

A user defines an SPL instance by selecting a set of features that satisfy her requirements. A generator composes the corresponding feature modules to yield a concrete list instance. Using the feature modules of Figure 4, we can generate a simple list using the base implementation only, but we can also use different combinations of the features (e.g., to derive a sorted synchronized list). Jak and FeatureC++ support static composition of classes. This means that according to the feature selection, the code of all refinements is composed into a single class at compile time.

Feature BASE

```
1  //Basic implementation of class List
2  template <class T>
3  class List {
4    Node<T>* head;
5    void add(T elem) { ... /* append at end */ }
6  };
```

Feature SORT

```
7  refines class List {
8    void add(T elem) { ... /* sorted insert */ }
9  };
```

Feature SYNC

```
10 refines class List {
11   SyncObject sync;
12   void add(T elem) {
13     LockObject lock(sync);  //syncronize access
14     super::Add(m);          //add element
15   }
16 };
```

Figure 4: FeatureC++ code of class `List` decomposed along the features Sort and Sync.

3.2 Component Hierarchies

We map the modeling concepts (i.e., named SPL instances, SPL specialization, and subtyping) to the implementation of SPLs by generating component hierarchies. Before we present implementation techniques, we review the requirements defined in Section 1 with respect to FOP.

Instance Identification. Indentifying different instances of the same SPL means to distinguish different variants of an *implementation class*[4] of these instances. For example, generating a sorted and a synchronized instance of ListPL results in different variants of class `List` (e.g., a sorted list in one instance and a synchronized list in the other instance; cf. Fig. 4). When using static composition mechanisms such as Jak, FeatureC++, or the C/C++ preprocessor, all variants of class `List` have the same name. This makes it impossible to identify the different variants of a class (e.g., for creating objects). Hence, the component generation process must create unique names for implementation classes of different SPL instances.

Subtyping. Mapping the specialization hierarchy to the implementation of an SPL means that generated components (i.e., the SPL instances) have to follow this hierarchy too. Hence, when an SPL `Base` is a super type of an SPL `Derived` then the whole set of implementation classes in `Base` should be a super type of the corresponding classes in `Derived`. This is also known as *Family Polymorphism* [8]. The resulting subtype relationship is needed to simplify client development and must be available in the client language. For example, FeatureC++ generates plain C++ code and we should be able to use a generated component in C++ clients. This requires a subtype relationship between implementation classes of different component variants in the generated C++ code. Since we can have different SPL spe-

[2]http://userweb.cs.utexas.edu/users/schwartz/
[3]http://fosd.de/fcc/

[4]We refer to the classes that are used to implement an SPL as *implementation classes*.

cializations for different application scenarios, the specialization hierarchy and thus the subtype relationship may be different for different client programs.

Code Reuse. Generating different variants of an SPL usually means code replication because the code of shared features is repeated for each instance. As a result, we get a similar increase in binary size as observed for C++ templates (a.k.a. *code bloat*). Since whole features are replicated between generated variants, the classes, methods, and refinements of a feature should be automatically reused across a family of SPL instances [14].

3.3 SPL Programming Interfaces

In Section 2.2, we defined the *semantic interface* of a (specialized) SPL. Based on this definition we define an SPL's *programming interface* as the union of the programming interfaces of the implementation classes defined in the features of the semantic interface. For example, the programming interface of `SortListPL` (cf. Fig. 3) consists of the interfaces of all classes defined in the base implementation of ListPL and feature Sort (cf. Fig. 1). It does not include classes or methods introduced by features QuickSort, MergeSort, or BubbleSort because only one of the features will be present in an instance. In specialization steps, we extend an SPL's programming interface up to a complete interface for a concrete component.

The subtype relationship between specialized SPLs also applies to the implementation classes. Hence, when SPL `Derived` is a specialization (and a subtype) of SPL `Base` then an implementation class `C` defined in `Derived` is also a subtype of class `C` defined in SPL `Base`. This means that a feature can only add members to the interface of a class but cannot modify members because it would conflict with the subtype relationship. For example, a feature cannot extend the signature of a method as it is sometimes done in preprocessor-based implementations of an SPL [12, 20]. We argue that such extensions must be avoided because they complicate SPL development and hamper use of SPL instances [20].

4. IMPLEMENTATION MECHANISMS

We present different mechanisms for generating component families that enable a programmer to use multiple instances of an SPL at the same time. We analyze each mechanism with respect to the presented requirements.

4.1 Namespaces / Packages

A simple way to distinguish sets of classes that have the same name is to group them into namespaces (C++) or packages (Java). The FeatureC++ compiler supports the generation of a package for each SPL instance. For example, we can use a namespace `SortList` to group all classes of a sorted list SPL instance. For Jak, this is possible with *refactoring feature modules* which move generated classes into a package [13]. Refactoring feature modules are a general mechanism that can be applied to other languages as well. In the following, we analyze the approach with respect to our requirements.

Instance Identification. We distinguish classes of different component variants via their package name. The package thus provides a unique type for each class. For example, we can define a client method that creates sorted lists:

```
class MailClient {
   sortList.List createList() {
      return new sortList.List();
   }
}
```

The name of package `sortList` corresponds to the name of the SPL instance defined in the MPL model (cf. Fig. 3). The instance name can thus be used for the code generation process. For example, the FeatureC++ compiler can generate required instances with their namespaces as defined in the composition model.

Subtyping. The namespace solution does not support subtyping between different variants of a class. The reason is that every generated SPL instance uses its own namespace and classes of one namespace are independent of the classes of a different namespace. For example, a class `quickSortList.List` is not a subtype of `sortList.List` even though both provide a similar interface. Furthermore, there is no representation of specialized SPLs in the namespace approach.

Code Reuse. There is no code reuse between classes of two SPL variants. For example, `quickSortList.List` and `mergeSortList.List` completely replicate the code of feature Sort and the base implementation. However, classes of different namespaces might be extracted and merged into a common class library [14]. This cannot avoid code replication completely but may be sufficient for many application scenarios. In contrast to the approach described in [14], static fields have to be handled differently to avoid shared state between different variants.

4.2 Virtual Classes

A namespace approach does not allow us to use implementation classes polymorhically. We can provide the required subtype relationship with *virtual classes* [15] as supported by CaesarJ[5] [1]. A virtual class is a nested class whose type depends on the type of an object of its enclosing class. In our case, the enclosing class represents a specialized SPL. With *mixin-based inheritance* [5], an enclosing CaesarJ class composes multiple classes. Mixin composition is similar to multiple inheritance but avoids some of its problems by linearizing the base classes. When implementing SPL features as enclosing classes, mixin composition can be used to compose features. The composition results in a specialized SPL that includes the features of all base classes. For example, in CaesarJ we define SortListPL (cf. Fig. 3) as follows[6]:

```
cclass SortListPL extends Sort & ListPL { }
```

`SortListPL` represents a specialized SPL that is defined via mixin composition of feature `Sort` with SPL `ListPL`.

Instance Identification. With virtual classes, a specialized SPL as well as an SPL instance is represented by a class. To use an SPL instance, we create an object of an SPL class (e.g., an instance of class `SortListPL`). The type of an implementation class, which is an inner virtual class, is defined by an object of an SPL class. For example, we can use an object of `SortListPL` to create sorted lists:

[5] http://caesarj.org
[6] CaesarJ classes are defined with keyword `cclass`.

```
cclass MailClient {
   SortListPL sortList = new QuickSortListPL();
   SortListPL.List createList() {
      return sortList.new List();
   }
}
```

In this example, `sortList` is an object of SPL instance `QuickSortListPL`. This specialized SPL has to correspond to a valid configuration. For example, we cannot create an instance of SortListPL because it does not provide a sorting implementation (cf. Fig. 3). The SPL instance object provides the `new` operator for creating objects of that instance. This is similar to the namespace approach. As in the namespace approach, we use the instance name `sortList` as defined in the MPL model.

Subtyping. The implementation classes of an SPL can be used polymorphically. For example, method

```
void display(SortListPL.List l)
```

accepts all kinds of sorted lists, which are defined in an SPL instance that is a subtype of `SortListPL`. Furthermore, due to the virtual class mechanism, a type can also be specified via an object. Hence, methods such as

```
void display(SortListPL plInst, plInst.List l)
```

can be used to ensure that an object (`plInst.List l`) corresponds to a particular SPL instance (`plInst`). In this example, the actual type of `plInst` could be `QuickSortListPL` (i.e., a subtype of `SortListPL`). List `l` then has to be an object of `QuickSortListPL.List`. This is used to distinguish objects of classes (e.g., `List`) of different SPL instances. With static type checking we can ensure that an object of one SPL instance is not passed to a different instance [9].

Code Reuse. The actual code reuse in a family of components depends on the concrete implementation of virtual classes. In CaesarJ, all implementation classes of a set of generated components form an inheritance hierarchy [1]. The hierarchy of a class corresponds to the refinement chain (i.e., the mixin list in CaesarJ) and is independent of the specialization hierarchy. This reduces code replication but does not completely avoid it: In a complex inheritance hierarchy, we have to replicate the code of refinements that are used multiple times at different positions in the hierarchy. However, this could be avoided with a different implementation.

Mixin Composition and Complexity Issues. The presented approach causes problems with respect to composition and complexity. The first issue is related to mixin composition. When creating a specialized SPL via mixin-based inheritance, we have to inherit from the SPLs as defined in our specialization hierarchy to achieve subtyping. At the same time, mixin composition is used to define the feature composition order: Features of the base classes are merged in the same order as they are listed in the base class definition. This entangles the subtype relationship and the feature composition order. Since feature composition is not commutative, it is impossible to achieve a valid feature order for all component hierarchies. A workaround in CaesarJ is to explicitly define the feature order as well as the parent SPLs required for subtyping. For example, we define a class `SyncSortListPL` (i.e., a synchronized sorted list), which should be a subtype of `SyncListPL` (cf. Fig 3) and `SortListPL` as:

```
cclass SyncSortListPL
   extends SyncListPL & SortListPL
         & Sync & Sort & ListBase { }
```

Here, `Sync`, `Sort`, and `ListBase` define the correct feature order; `SyncListPL` and `SortListPL` are used to define the required subtyping. However, mixin composition still increases the complexity of the SPL configuration process, which hinders its use for SPL development:

- Repeating the feature order for every SPL instance means additional configuration effort and is error-prone.
- An SPL instance is created by a user of an SPL (e.g., a developer of a client application) that does not know SPL implementation details such as the feature composition order.
- Changing the configuration of an inner component of the hierarchy (e.g., adding a feature to `SortListPL`; cf. Fig 3) is not possible without modifying every instance to explicitly define the feature order.
- The approach imposes an additional complexity on client developers due to the use of virtual classes. Hence, a client developer that uses only a single SPL instance is faced with an unneeded complexity.

Some of these issues can be solved by extending CaesarJ, e.g., by separating composition order from subclassing. We propose to address these complexity issues with a generative approach: based on an SPL implementation in an FOP language such as Jak, we generate virtual classes (e.g., CaesarJ code) including the specialization hierarchy with mixin-based inheritance. This generative approach avoids manual configuration via mixin composition. Furthermore, when only a single SPL instance is required, we generate plain Java code and avoid the complexity of virtual classes.

4.3 Generating SPL Interfaces

Both, the namespace approach and the virtual class approach, have drawbacks that limit their applicability. For a more general solution, we generate a hierarchy of SPL interfaces (i.e., the SPL's programming interface) to represent specialized SPLs. We use nested interfaces to represent the interfaces of SPL implementation classes. In Figure 5, we show an example for the generated interfaces of a subset of the ListPL hierarchy (Lines 1–15). A concrete SPL instance is defined as a class that implements the interface of the corresponding specialized SPL (Lines 16–20). This code transformation is similar to the implementation used in CaesarJ [1].

In contrast to the namespace approach, implementation classes of an instance are defined as nested classes within their instance (Lines 18–19). In contrast to virtual classes, the SPL specialization hierarchy is represented as a hierarchy of interfaces in the client language. We thus separate an SPL instance (i.e., a concrete implementation) from its interface. This corresponds to the fact that we can have an SPL specialization that does not correspond to a concrete instance. The interface can be used to define which functionality an SPL provides without a concrete implementation.

The generated interface hierarchy provides an emulation of virtual classes for plain Java and C++. As implemen-

SPL ListPL

```
1  interface ListPL {
2   abstract List newList();
3   interface List {...}
4   interface Node {...}
5  }
```

SPL specialization SortListPL

```
6  interface SortListPL extends ListPL {
7   abstract List newList();
8   interface List extends ListPL.List {...}
9   interface Node extends ListPL.Node{...}
10 }
```

SPL specialization QuickSortListPL

```
11 interface QuickSortListPL extends SortListPL {
12  abstract List newList();
13  interface List extends ListPL.List {...}
14  interface Node extends ListPL.Node{...}
15 }
```

SPL instance QuickSortList

```
16 class QuickSortList implements QuickSortListPL {
17  List newList() {...}
18  class List implements QuickSortListPL.List {...}
19  class Node implements QuickSortListPL.Node{...}
20 }
```

Figure 5: Generated interface hierarchy representing SPL specialization (Lines 1–15) and a generated class representing an SPL instance (Lines 16–20).

tation mechanism, it is possible to use refactoring feature modules to extract the required interface of a specialized SPL. In the following, we analyze this solution with respect to the requirements defined in Section 1.

Instance Identification. We refer to an implementation class via its enclosing SPL class (which represents a concrete instance) or indirectly by using a generated factory method (e.g., method `newList()` in Fig. 5), which is part of the generated interface:

```
class MailClient {
  SortListPL sortList = new QuickSortList();
  SortListPL.List createList() {
     return sortList.newList();
  }
}
```

In this example, `createList()` invokes the virtual factory method `newList()` (cf. Fig. 5), which is implemented by an SPL instance that is a subtype of `SortListPL`. This is similar to the use of the `new` operator in virtual classes which simplifies to write generic client code.

In contrast to virtual classes, an SPL instance is represented by a class that implements the interface of a specialized SPL. This means that we can have different implementations of the same (fully) specialized SPL (i.e., that implement the same interface). For example, we can have two sorted lists, one providing a speed optimized implementation and one providing a memory optimized implementation and both can be used polymorphically. It is also possible to implement the same interface in two different SPLs, which allows us to use instances of the SPLs interchangeably. Hence, we extend the interface concept of components to component SPLs. This is different from the virtual class solution (cf. Section 4.2) where we cannot distinguish specialized SPLs from SPL instances.

Subtyping. Subtyping of SPL implementation classes is realized as subtyping between the nested interfaces. As with virtual classes, the interface of a class can thus be used polymorphically. We can use it to reference all variants of a class that are defined in a subytpe of the specialized SPL. For example, a client method

```
void display(SortListPL.List l) { ... }
```

accepts all kinds of sorted lists. Similar to the CaesarJ approach, this solution achieves static subtyping also in case of multiple inheritance between SPL specializations. It is implemented as multiple inheritance between the SPL interfaces and the nested interfaces of implementation classes. For example, a sorted and synchronized list can be a subtype of `SortListPL` and `SyncListPL`. This does not cause problems known from multiple inheritance of implementation classes. Finally, the approach allows us to add new SPL specializations without modifying a client implementation as long as the new variant is not a super type of an existing one (i.e., it does not modify the existing inheritance hierarchy).

The main drawback of this solution is that it does not allow us to statically check whether two objects of implementation classes are compatible with each other (i.e., if they are part of the same SPL instance). For example, using the abstract list interface, we could pass a node of a single linked list to a double linked list causing a runtime type error. With virtual classes, such errors can already be detected by the compiler [9].

Code Reuse. In the presented solution we do not address the problem of code replication. Even though we are using nested interfaces to represent a specialized SPL we cannot use nested classes to represent fragments of SPL implementation classes. It would result in the same problems as observed for mixin composition (entangled feature order and subtyping). Furthermore, it would result in multiple inheritance of implementation classes when multiple inheritance of their interfaces is needed. For example, a sorted and synchronized list would inherit the basic list implementation twice. However, a similar implementation as used in CaesarJ could be used to avoid code replication.

5. DISCUSSION

We presented different approaches to generate component families that allow us to use multiple variants of a component in the same program. In the following, we discuss open issues and suggest how FOP approaches should be changed to provide a viable solution for generating components from an SPL.

5.1 Code Reuse

In CaesarJ, code replication is reduced. It can be completely avoided with an implementation that avoids replication of refinements, e.g., using delegation. This is also possible for generated OO hierarchies and even for the namespace approach. However, it means a more complex code transformation than simply adding a namespace. Furthermore, it may result in an overhead in terms of execution time for introduced indirections, which has to be evaluated.

5.2 When to Use Which Mechanism?

Since all presented solutions have benefits and drawbacks, no mechanism can be generally preferred. We discuss when the different mechanisms should be used.

Plain Static Composition. When using a component, most of the time this means to use a single instance of the component only. This can be accomplished with current approaches for SPL development that use static composition of features. Furthermore, the code transformations used for Jak and FeatureC++ allow us to use generated components in plain Java and C++ clients.

Namespaces. The namespace approach often suffices when a client uses multiple variants of a component. However, it does not support subtyping of generated components. This causes a high effort to write generic code for different component variants. Compared to the advanced solutions that support subtyping, the namespace approach achieves better performance due to the possibility of method inlining. However, a detailed performance evaluation is needed to analyze the actual effect. Due to its simplicity, it can also be used on deeply embedded devices when there is no support for OOP or for some OOP concepts such as virtual methods.

Virtual Classes. Implementing an SPL with virtual classes (e.g., with CaesarJ) allows us to to achieve subtyping of a component hierarchy. By generating virtual classes from an FOP implementation, we avoid the complexity of mixin composition. However, once we have decided for such an SPL implementation, a client developer is faced with the complexity of virtual classes even when not needed. A remaining problem is that we cannot use this solution when the client is developed with a mainstream OO language due to missing support for virtual classes.

Generating SPL Interfaces. To achieve subtyping between SPL specializations in languages that do not support virtual classes, we propose to generate plain OO interfaces to represent specialized SPLs. This allows us to access different variants of a class with the same interface. The approach also allows us to separate SPL instances (implemented as classes) from SPL specializations (implemented as interfaces) and to have different implementations of the same SPL specialization. The main drawback compared to virtual classes is that we loose parts of static type safety on the client side.

5.3 Flexible Feature Composition

To allow programmers to choose the best solution according to the application scenario, we propose to use a flexible approach for feature composition that generates the actually required code. We already support this for plain static composition and generating namespaces / packages. When a component hierarchy and subtyping is needed, we propose to use more advanced approaches: generating virtual classes if supported by the client language or generating a plain OO interface hierarchy otherwise. It is also possible to extend CaesarJ to avoid the problems mentioned above. From CaesarJ code we could then generate code without virtual classes when they are not needed or not supported.

A flexible approach allows us to switch from one implementation mechanism to another by regenerating the components (i.e., when the requirements change). However, this also means that the client, which uses the SPL, has to be changed accordingly. With refactoring feature modules we can automatically refactor the client program as well; but this has to be further analyzed in future work.

6. RELATED WORK

There are also other languages that support virtual classes, which we could have used for our analysis. However, we think that the problems are very similar to those described for CaesarJ.

Nested Intersection. The language J& supports composition of multiple components using nested intersection [17]. It is based on composition of classes and packages with their inner classes similar to virtual classes. J& might be better suited for implementing specialization hierarchies than virtual classes because it defines *static* virtual types, which are attributes of packages or classes and not of objects. However, the composition mechanism does not linearize class extensions, which complicates development of independently composable features. We intend to evaluate the approach for implementing specialization hierarchies in further work.

Mixin Layers. Generics, such as C++ templates, can be used to implement layered designs [22]. Similar to virtual classes, nested classes of a mixin layer extend classes of their super layers. As a precursor of FOP, the language P++ (an extension of C++) provides composition of mixin layers and explicitly defines layer interfaces [2]. Static mixin composition is similar to mixin composition of virtual classes but different instances of a component are generated by template instantiation at compile time. However, as in virtual classes, the feature composition order and subtyping are not independent. Moreover, generating hierarchies with multiple inheritance would result in multiple inheritance of inner classes. Jiazzi solves some of the problems of static mixin composition with concepts similar to virtual classes [16].

Dynamic Feature Composition. Dynamic composition of features means to derive an SPL instance by composing features at runtime (e.g., supported by Delegation Layers [18], Object Teams [10], and FeatureC++). Delegation layers and Object Teams furthermore combine delegation-based composition with virtual classes. Dynamic composition provides more flexibility than static composition of features because the feature selection of an SPL instance is determined in a running program. When this flexibility is needed we do not want to statically define a specialization hierarchy as proposed in this paper.

7. CONCLUSION

Feature-oriented software development lacks support for reusing multiple products of an SPL in the same program. For example, programmers cannot model or implement large software systems that use multiple component variants generated from an SPL. We propose to model and generate *component hierarchies* from a feature-oriented SPL. A component hierarchy allows a programmer to distinguish different variants of a component and provides a subtype relationship between components. This enables client developers to write generic code to be used with different variants of a component.

Based on modeling support for component hierarchies, we apply the concept to feature-based software composition. Since a component hierarchy is only needed when using different variants of the same component, we propose to use a flexible approach to feature composition:

- we use plain static composition if only a single instance of an SPL is used at a time,
- we generate namespaces when using multiple compo-

nent variants at the same time,

- we propose to generate component hierarchies when subtyping is needed: (1) by generating virtual classes or (2) by generating a hierarchy of OO interfaces when the client language does not support virtual classes.

Due to a flexible composition mechanism a developer of a component SPL does not have to consider the special needs of different clients. Based on a feature-oriented implementation of an SPL, a client developer defines the required component hierarchy (or uses a predefined one) and generates the components that correspond to the application scenario and the client language.

In future work, we plan to implement and evaluate the proposed solutions for generating component hierarchies. This means to connect modeling of component hierarchies and feature composition and to extend the FOP code generation process accordingly (e.g., for FeatureC++).

Acknowledgments

We thank Don Batory for discussions about the presented work. Marko Rosenmüller is funded by German Research Foundation (DFG), project number SA 465/34-1.[7] Norbert Siegmund is funded by German Ministry of Education and Research (BMBF), project number 01IM08003C.[8]

8. REFERENCES

[1] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An Overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *LNCS*, pages 135–173. Springer, 2006.

[2] D. Batory, S. Dasari, B. Geraci, V. Singhal, M. Sirkin, and J. Thomas. Achieving reuse with software system generators. *IEEE Software*, pages 89–94, 1995.

[3] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Softw. Eng. (TSE)*, 30(6):355–371, 2004.

[4] D. Batory, V. Singhal, M. Sirkin, and J. Thomas. Scalable software libraries. *SIGSOFT Softw. Eng. Notes*, 18(5):191–199, 1993.

[5] G. Bracha and W. R. Cook. Mixin-Based Inheritance. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) and the Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 303–311. ACM Press, 1990.

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

[7] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged Configuration Using Feature Models. In *Proc. Int'l. Software Product Line Conf. (SPLC)*, volume 3154 of *LNCS*, pages 266–283. Springer, 2004.

[8] E. Ernst. Family Polymorphism. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 2072 of *LNCS*, pages 303–326. Springer, 2001.

[9] E. Ernst, K. Ostermann, and W. R. Cook. A Virtual Class Calculus. In *Proc. Int'l. Symposium on Principles of Programming Languages (POPL)*, pages 270–282. ACM Press, 2006.

[7] http://fosd.de/multiple

[8] http://vierfores.de

[10] S. Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *Proc. Int'l. Net.ObjectDays Conf.*, volume 2591 of *LNCS*, pages 248–264. Springer, 2002.

[11] 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.

[12] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proc. Int'l. Conf. Software Engineering (ICSE)*, pages 311–320. ACM Press, 2008.

[13] M. Kuhlemann, D. Batory, and S. Apel. Refactoring Feature Modules. In *Proc. Int'l. Conf. Software Reuse (ICSR)*, pages 106–115. Springer, 2009.

[14] J. Liu and D. Batory. Automatic Remodularization and Optimized Synthesis of Product-Families. In *Proc. Int'l. Conf. Generative Programming and Component Eng. (GPCE)*, pages 379–395. Springer, 2004.

[15] O. L. Madsen and B. Moller-Pedersen. Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 397–406. ACM Press, 1989.

[16] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New-Age Components for Old-Fashioned Java. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 211–222. ACM Press, 2001.

[17] N. Nystrom, X. Qi, and A. C. Myers. J&: Nested Intersection for Scalable Software Composition. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 21–35. ACM Press, 2006.

[18] K. Ostermann. Dynamically Composable Collaborations with Delegation Layers. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 2374 of *LNCS*, pages 89–110. Springer, 2002.

[19] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 419–443. Springer, 1997.

[20] M. Rosenmüller, M. Kuhlemann, N. Siegmund, and H. Schirmeier. Avoiding Variability of Method Signatures in Software Product Lines: A Case Study. In *Workshop on Aspect-Oriented Product Line Engineering*, pages 20–25, 2007.

[21] M. Rosenmüller and N. Siegmund. Automating the Configuration of Multi Software Product Lines. In *Proc. Int'l. Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 123–130, 2010.

[22] Y. Smaragdakis and D. Batory. Implementing Layered Designs with Mixin Layers. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 1445 of *LNCS*, pages 550–570, 1998.

[23] T. Thüm, D. Batory, and C. Kästner. Reasoning about Edits to Feature Models. In *Proc. 31th Int'l. Conf. Software Engineering (ICSE)*, pages 254–264. IEEE CS, 2009.