Abstract

We have observed that large software systems are increasingly defined in terms of the features they implement. Consequently, there is a need to express the commonalities and variability between products of a product line in terms of features. Unfortunately, technology support for this is currently limited to the requirements level. There is a need to extend this support to the design and implementation level as well. Existing technologies such as AOP and SOP may be of use here. However, features are not first class citizens in these paradigms.

To address this and to explore the problems and issues with respect to feature composition at the implementation level, we have formalized the notion of features and the composition of features. In addition we have specified an algorithm for mapping features to Java classes. Using our model and composition algorithm, we can select a number of base components and a number of features from a software product line and derive a product. As a proof of concept we have experimented extensively with a partial Java implementation of our approach.

1 Introduction

Software applications grow larger and larger, are maintained for longer periods of time and need to be updated frequently to evolve with new needs and changing consumer requirements. To cope with this increasing size of software applications a Software Product Line (SPL) [4] approach can be used. A SPL is designed for a family of (domain) related applications. It consists of a product-line architecture and a set of reusable components. Specific applications may be derived from the SPL by selecting, enhancing and adapting components. We have observed that, during product derivation, the differences between products are usually defined in terms of features [14][15][3][8].

The derivation process is an expensive process. The reason for this is that there is a mismatch between the way products are defined (i.e. in terms of features) and the variability offered by the SPL. Requirements changes generally result in changing and/or adding features to the SPL. Features have been identified as optional or incremental units of change [7]. However, typically features have no first class representation in the SPL implementation. During product derivation, developers must make adaptations to the SPL's provided feature set in order to implement product specific features. Consequently, changing the implementation in order to meet new requirements is potentially expensive because code related to one feature may be spread over multiple software components.

Ideally, new or changed features would be captured in separate pieces of code that can be changed and maintained independently. Thus changes during the product derivation would be limited to those pieces of code. The main topic of this paper is giving features a first class representation in SPLs so that during product derivation, product developers may select features from the SPL and reuse them in their products. There are a number of problems associated with this type of product derivation. A key contribution of this paper is that we identify those problems and demonstrate in our approach how these can be worked around or solved.

We use a top down approach of analyzing the issue of feature based product derivation. Our top-down approach starts with the modeling of concepts such as features and SPLs in terms of sets. With the help of this formal description of the feature model composition problems are identified. Several solutions to these composition problems are presented. One of these solutions is used in a prototype implementation that is also presented in this paper. The three contributions of this paper are:

• A feature model modeling features.
• A classification of feature composition problems and potential solutions to these problems.
• A demonstration how features can be realized at the implementation level. This opens the way to automatically derive a product from a SPL based on a selection of available features.

1.1 Overview of the paper

The remainder of this paper is organized as follows. In Section 2, an informal description of our model is presented. Section 3 presents a formal notation to define the se-
ments of feature composition. This formal notation is used in Section 4 to derive a classification of composition problems and potential solutions. In Section 5 a composition algorithm for a java based prototype is discussed. An overview of related work is presented in Section 6. Finally, the paper is concluded in Section 7.

2 Features in SPLs

To clarify our approach, an examination of how features and SPLs are related is presented. After this, an informal outline of our approach is presented. The approach involves features, actors and roles. An analogy with the Hollywood movie industry is used to illustrate and clarify these terms. At the end of the section the approach is exemplified using the example of a video shop renting system.

2.1 Software Product Lines (SPLs)

A Software Product Line (SPL) consists of a base implementation (e.g. B) and a number of features (e.g. \( F_1 \ldots F_{25} \)). A product may be derived from the SPL by selecting an arbitrary number of these features and combining these with the base implementation (e.g. \( B + F_9 + F_{18} + \ldots + F_{23} \)). The base implementation itself can also be seen as a set of (standard) features, i.e. an SPL then becomes for example \( F_1 + F_2 + F_3 + F_4 + F_9 + F_{18} + \ldots + F_{23} \), where some features (e.g. 1 to 4) are standard features (in FODA these are called mandatory features [14]) and others are optional features. In this paper base components model entities, which cannot easily be decomposed into features. Legacy code components and domain components are examples of these base-components.

The properties of the composition-operator + that is used in the composition is our primary interest in this paper. Of course, it would be ideal if that operator was associative (i.e. \((F_1 + F_2) + F_3 = F_1 + (F_2 + F_3)\)) and commutative (i.e. \(F_1 + F_2 = F_2 + F_1\)). Then, a product developer would be able to arbitrarily combine features. The developer of each feature would not need to worry about interaction with other features. This way it would not make any difference at what point in time and/or development a certain feature is brought into a feature composition. However, in general (as will be argued in the next section) this operator is neither associative nor commutative, because of feature dependencies: one feature may depend on another feature. This is for example the case if one feature cannot operate without another feature.

Things are even worse in the sense that composition of features might introduce feature interaction [7]: a feature interaction is some way in which one or more features modify or influence another feature in describing the system’s behavior set. Due to feature interaction, composition of features might become incomplete, inconsistent, non-deterministic, unimplementable, etc. (see [27]). The method introduced in this paper keeps the composition complete, consistent, deterministic and implementable.

2.2 Roles, actors and base-components

In order to illustrate how features can be modeled and what kind of relationship they have with the base-components, we use an analogy with the Hollywood movie industry. Our application can be considered as a movie with actors playing one or more roles. The movie under consideration consists of a number of scenes. Choosing different sets of scenes results in different movies. For example, a shortened version for airlines or a director’s cut for DVD, each with a different set of scenes. Within each scene of a movie production, actors play one or more roles. Normally, one role cannot be changed, because a number of roles will be dependent on each other, hence the scenes will change, resulting in a different movie.

So, a movie consists of a set of scenes that implement a script. By selecting a number of scenes a certain movie can be created. Each scene features a number of actors that play different roles. An actor plays a role, possibly more than one. For example, Kevin Kline plays multiple roles in the movie “Fierce Creatures” (1997). Also more then one actor can play the same role. For example in Hollywood productions the use of stand-ins is common to replace expensive actors in dangerous scenes. In soap series it isn’t uncommon that a role is taken over by another actor.

<table>
<thead>
<tr>
<th>Model</th>
<th>Analogy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPL</td>
<td>All, actors, scenes and roles</td>
<td>A SPL captures the commonalities between the various products that are derived from it. In Hollywood terminology this might be the raw material the movie director works with when producing a movie.</td>
</tr>
<tr>
<td>Product</td>
<td>Movie</td>
<td>A product is derived from a SPL similar to how a director produces a movie.</td>
</tr>
<tr>
<td>Feature</td>
<td>Scene</td>
<td>A feature is part of one or more products, just like a scene can be part of one or more movies. For example it is not uncommon to have different versions (student, professional, enterprise) of the same application, they only differ in the features they support.</td>
</tr>
<tr>
<td>Base-component</td>
<td>Member of the cast</td>
<td>In our composition model, features are associated with base components just like in a movie roles are played by members of the cast.</td>
</tr>
<tr>
<td>Roles</td>
<td>Roles</td>
<td>In the model roles are part of features, as in the Hollywood analogy roles are part of a scene.</td>
</tr>
</tbody>
</table>

Table 1: Feature model elements versus Hollywood analogy terms
Now that the analogy is introduced, the elements of the feature model can be presented. Table 1 lists the various elements of the feature model and their Hollywood analogy counterparts. Figure 1 visualizes the feature model elements and their relationships. The base-components visualized in the top part of Figure 1 are entities, which cannot easily be decomposed into features and belong to the base SPL implementation. On the left side of Figure 1 two features containing one and two roles are presented, visualizing the fact that roles are part of a feature. At the center there are four actors.

The top two actors consist of base-components 1 and 2, both playing role 1. The bottom two actors are base-component 1 playing role 2 and base-component 2 playing role 3. At the bottom of Figure 1 the derived components are situated. A derived component is a base-component incorporating actors playing the roles of the selected features.

The concepts we discussed here form the basis of our composition approach, which will be elaborated on in Section 3. Before that, however, we provide an example.

![Feature Model Diagram](attachment:image.png)

**Figure 1** a conceptual view of the feature model

### 2.3 Case

Throughout the rest of this paper a video renting administration system is used to illustrate various aspects of the feature model. A quick domain analysis provides the following domain components for the video-shop system: a VideoShop component, a Video component and a Customer component. These three domain components are the base-components of this case. For the remaining part of the paper components are typeset in **bold**. Features are typeset underlined. The following features have been selected for this case:

- **VideoRental**: A **Customer** can rent a **Video**
- **ReturnVideo**: A **Customer** can return a **Video** that is rented.
- **AmountDiscount**: A **Customer** receives a certain discount when renting more then one **Video**
- **RegularCustomerDiscount**: A regular **Customer** receives a certain discount when renting a **Video**
- **AgeControl**: Only a **Customer** above a certain age may rent a certain **Video**

These features are selected because they illustrate the various issues of the feature model. The system should always contain the features VideoRental and ReturnVideo, for the system to have a minimal of functionality. Both features, however, will not be part of the base components because the specification of the features might change over time. The other features are optional. Some features are dependent on each other, e.g., all optional features depend on VideoRental. Also, some features will have feature interaction, for example AmountDiscount and RegularCustomerDiscount, both influence the amount of money a customer has to spend.

Figure 2 presents an overview of the video-shop case. The different features of the video-shop, e.g. VideoRental, ReturnVideo, etc., can be found on the left side. The base-components (VideoShop, Customer, Video) are found on the top. The features consist of one or more roles, for example the VideoRental feature consists of the following roles (roles are in teletype): Contract, Contract-Maintenance, RentedItem, Renter and Maintenance.

The first role, Contract of the VideoRental feature, introduces a new concept into the feature model. The Contract role introduces a new component in the composition, the **Contract**, which isn’t directly related to any existing base-component. New concepts in the domain may be added by roles defining new components.

Because features are decomposed further into roles, the core of our composition method consists of mapping the defined roles onto the base-components. The mapping of the different features and roles is visualized in Figure 2. For each of the features the roles are presented and the functionality of the roles is visualized by displaying the signatures of the corresponding methods. Furthermore the functionality of the base-components is visualized.

When a role is mapped onto a base-component an intermediate component is created (we refer to these intermediate components as actors), these are the little rectangles in the middle. Each actor has a unique name, for example the name of the **Contract** actor is A1-4. The first number indicates that the actor belongs to the first feature (i.e. VideoRental). The second number indicates the base-component to which the role has been mapped. The **Contract** role has been mapped to a new 4th base-component.

### 2.4 Summary

This section presents the concepts used in our feature composition model. Using a Hollywood analogy various concepts was introduced. The feature model models the relationships between features and the components of a Software Product Line. The various feature model elements, such as: features, actors, base-components, derived components and roles, are described and their relationships explained.
3 The formal model

Now that the feature model has been introduced and has been illustrated by the video-shop case, the mapping of the roles and base-components onto the actors remains. The composition of roles, actors and base-components and actors, resulting from the mappings onto roles, proves to be far from trivial. In this section, we formalize the notion of features and feature composition in order to be able to identify potential composition problems. The feature model is formally defined in terms of sets. Mappings of elements (e.g. ) of one set to elements of another set denote relationships between those elements.

In the model a method signature is denoted by an interface, a unique identifier for the complete definition of the method itself without an implementation, for example in Java this is the header of a method, including the method name, the list of parameters, and the type of the returned value. A set of such signatures is called an interface:

\[
\text{interface} = \{\text{operationSignature}_{i} | i \in \text{signatureSet}\}
\]

With this notation an interface is denoted as a set of operation signatures, named operationSignature_{1}, operationSignature_{2}, etc., where signatureSet is the complete set of the available operation signatures.

A role is a set of interfaces and a one to one mapping of the operation signatures of the interfaces to implementations of these operation signatures. In imperative languages this implementation can be seen as a code block, i.e. the body of a method without the header.

A role can now be defined as:

\[
\text{role} = \{\text{interface}_{k} | k \in \text{interfaceSet}\}, \{\text{operationSignature}_{k_{i}} \rightarrow \text{implementation}_{k_{i}} | k_{i} \in \text{interfaceSet} \land i \in \text{signatureSet}_{k}\}
\]

The mapping describes that an operation signature is implemented by associating an implementation with the operation signature. A role is a partial implementation, mapped onto a component. Separate roles in a feature are required to model the fact that one component can have multiple roles in the context of a feature. To do the mapping of a role onto a base-component an intermediate form may be used. In a feature the implementations are mapped onto actors. An actor is a set of roles from a feature, mapped to a base-component. An actor can be seen as an intermediate component.

A feature is a set of roles, a set of actors, and a many to many mapping from roles to actors, i.e.:

\[
\text{feature} = \{\text{role}_{i} | r \in \text{roleSet}\}, \{\text{actor}_{o} | o \in \text{actorSet}\}, \{\text{role}_{i} \rightarrow \text{actor}_{j} | i \in \text{roleSet} \land j \in \text{actorSet}\}\}
\]

A role may be mapped to more than one actor. Also, more then one role can be mapped to the same actor. One role can map to more than one actor, if the corresponding code is going to be used in more than one component. Although this will in general be considered a signal of bad design, it is not excluded in our model.
A software product line (SPL) consists of all features and all base-components:

\[
\text{SPL} = \\
\{ \text{feature} | f \in \text{featureSet} \} \cup \\
\{ \text{baseComponent} | o \in \text{baseSet} \}
\]

A specific product, derived from an SPL, consists of a selected number of features, a set of derived components and the mapping from the actors to the derived component implementations, which in turn are derived from the base-components, i.e.:

\[
\text{product} = \\
\{ \text{feature} | s \in \text{selectedFeatures} \subseteq \text{featureSet} \}, \\
\{ \text{component} | (c \in \text{compSet}) \}, \\
\{ \text{actor} \rightarrow \text{component} | i \in \text{actorSet} \subseteq \text{compSet} \}
\]

The set of derived components is derived from the set of base-components, through the mapping of the actors to the base-components. Therefore, the set of derived components contains at least as many elements as the set of base-components, i.e.:

\[
\{ \text{baseComponent} \} \subseteq \{ \text{components} \}
\]

The transformation from a base-component to a derived component is not formalized here. This transformation is the main issue in our approach and is investigated further in the following sections. Our model currently does not model feature dependencies. Such dependencies further complicate the composition process. However, they are not relevant for the identification of potential feature composition problems that are discussed in the next section.

Figure 3 illustrates our approach: methods are mapped onto the components, through the actors. Actors may introduce new components, which are independent of the defined base-components. These new components are dependent on an additional, initially empty, base-component `none`.

Note that we have introduced three types of components: the base-components, new components, and derived components. The base-components come from a domain model or are legacy components. The new components are components introduced by the roles of new features. The derived components are components generated for a specific derivation.

### 3.1 Summary

This section presents a formalization of the feature model earlier proposed. Features, roles, base-components, new components, and derived components were introduced and formalized. In the next section the used composition is investigated.

### 4 The Composition Operator

In this section, the composition operator for the feature model is investigated. The composition operator in the feature model is used to compose features with base-components. A feature, however, consists of one or more roles that map onto an actor. This section investigates how an Actor can be composed of roles and base-components, what the associated composition problems are and how these problems may be solved.
4.1 Introduction

An actor consists of roles and base-components. In the feature model, the derived components contain the functionality of the corresponding base-components and actors of the selected features that are mapped to the base component. As mentioned earlier, feature dependencies complicate the composition process. Ideally the order in which features are mapped to base components would not affect the semantics of the derived components. However, because of the dependencies the order does matter (i.e. the composition operator is not commutative). Conceptually, the actors are accumulated on top of the base components (i.e. each actor is composed with the composition of all previous actors and the base components). Each actor combines various roles of a feature that are mapped to the same base component.

For example, Figure 4 visualizes the composition of two roles (R1 and R2). To simplify the composition problem, Figure 4 does take into account that an actor can be composed with a base-component or another actor. However, this has no consequences for composition operator. Both roles (R1 and R2) consist of one operation, denoted by $S_1$ and $S_2$, and an implementation for this operation (i.e. $I_1$ and $I_2$). The actor that results from the composition of role A and B should contain the unified behavior of roles A and B. Both implementations A and B are considered to be black boxes. The composition of the actor then becomes the problem of “gluing” both implementations together, as denoted in Figure 4 with the question marks.

4.2 Analyzing the Composition of Roles

Both operation signature and implementation have an effect on the “glue” that is needed to compose the implementations. By looking at the relationships between the operation signatures and the implementations, four different types of composition can be identified:

1. Signatures and implementations are all different. Figure 3 illustrates this: roles R2 (with $S_2 \rightarrow I_1$) and R6 (with $S_6 \rightarrow I_2$). R2 is mapped onto actor A1-2 and R6 is mapped onto actor A3-2. Both A1-2 and A3-2 are mapped onto the same base-component BC2. An example in the video shop (Figure 2) is the Renter and Returning roles. This situation does not raise any problems because there is no interaction between the implementations.
2. The signatures are different and the implementations are equal. In Figure 3 this is illustrated in roles R2 (with $S_2 \rightarrow I_2$) and R4 (with $S_4 \rightarrow I_1$). Role R2 is mapped onto actor A1-2 and R4 onto A2-2. Both A1-2 and A2-2 are mapped onto the same base-component BC2. The video shop example does not contain this situation.
   This situation does not present any problems either. It might signal bad design because different signatures can be implemented using the same implementation so the signatures might be considered equal instead of different.
3. Both the signatures and implementations are equal. This looks like copy-paste reuse, also a bad practice. In Figure 3 this is illustrated in roles R1 (with $S_1 \rightarrow I_1$) and R7 (again $S_7 \rightarrow I_7$). R1 is mapped onto actor A1-4, R7 onto A3-4 and both actors are mapped onto the same base-component BC4. The video shop example does not contain this situation.
   Although code fragments appear double in the resulting application there are no serious problems. Problems may arise, however, when maintenance is needed (code needs to be repaired in different places). A simple solution for this kind of problems is simply mapping the different code fragments to just one fragment.
4. The signatures are equal and the implementations are different. In Figure 3 this is illustrated in roles R4 (with $S_4 \rightarrow I_4$) and R6 (with $S_4 \rightarrow I_6$). Role R4 is mapped onto actor A2-2, R6 onto actor A3-2 and both actors are mapped onto the same base-component BC2. In the video shop case an example of this situation can be found with the operation rents in the roles Renter and AmountDiscount. This is a serious problem that requires further investigation. The remainder of this section is devoted to this problem.

Of the four described combinations for composing the roles, only the last one is problematic. The combination of one signature with more than one implementation can be illustrated best with Lego-building blocks. An operation signature is the header and the implementation the body of a method. Equal operation signatures therefore means that the headers are equal, thus the parameter list and return type of the methods implementing the operation signature are equal, only the body is different.

A Lego-brick can be seen as a shape representing the operation signature: the shape of the top of the brick illustrates the parameter list and the shape of the bottom of the brick illustrates the return type. The inside of the brick represents the implementation. Because in situation four, the signature is the same for both implementations, the Lego bricks have the same shape. This results in three ways to combine the implementations, as illustrated in Figure 5:

A. No input parameters, no output. The operation signature of the implementations has no return type and no parameters. This is the easiest situation because the implementations may just be concatenated.

B. Input is equal to output. If the implementations have the same input type as the return type, the implementations may just be piped together. However, this requires that the semantics of the input and output match.

C. Input and output are different. In this situation the implementations can neither be concatenated nor piped.
There are a number of issues with the different combination strategies described above:

- **Scope of variables.** Both implementations may have a common set of local variables with different semantics, which may raise some conflicts when both implementations are combined. A possible solution is to automatically rename conflicting local variables.

- **Side effects.** Both implementations may have conflicting side effects. For example, both implementations may throw an exception. The code of the second implementation may never be executed if the first implementation throws an exception. There are many subtle ways both implementations may conflict which need to be considered when combining the implementations.

It should be pointed out that other approaches (e.g., AspectJ [16][17]) exhibit the same sort of problems. Especially AspectJ has become a complex language due to the fact that it tries to solve/work around these issues.

### 4.4 Summary

In this section the formal model of the feature composition model has been used to examine where exactly this composition becomes problematic, namely when combining implementations with the same signature into one implementation. We have outlined three strategies that may be combined, for doing so. However, there are a number of issues that prevents a universal solution to this problem. Any implementation of our feature composition model requires that these issues be addressed in some way. In the next section we will outline the composition algorithm we used to implement the video shop prototype.

## 5 A prototype implementation of the feature model

In this section a prototype implementation of the feature model is presented. The implementation has been implemented in the object-oriented programming language Java. The prototype, the implementation choices and potential alternative solutions are presented. The section ends with an overview of implementation issues and their solutions.

### 5.1 The prototype

As explained in Section 4, there are three ways to compose role implementations. There are two strategies for selecting the composition method that should be used:

- **Explicit Composition.** Explicit composition means that the developer explicitly defines how the composition should take place. In addition to specifying which roles map to which base-component, the developer also specifies how to compose the roles. The disadvantage of this approach is that this complicates the composition process.

- **Implicit Composition.** With implicit composition the rules for the composition process are pre-defined and composition is done according to these rules. The
dependency between roles of the same feature. An example of dependencies among roles in the same feature is in the Contract role and the necessary Contract operations in the Customer and Video (these are in the ContractMaintenance role) are known.

- Dependency between roles of different features. Dependencies between roles of different features are the basis for feature dependencies. Feature dependencies in the feature model are the result of roles depending on roles or actors of a different feature. An example is the role ExpiredContract of ReturnVideo, which is dependent on the VideoRent feature, because of the needed concept of a Contract. This concept is introduced in the VideoRental feature by the Contract role.

- Dependency between a role and an actor. This dependency is different from a role depending on another role, because in this dependency a role is dependent on the composition of a role and a specific base-component, which is an actor. An example is the Return role of the ReturnVideo feature. The Contract that the Returning role uses should have the Contract role (from the VideoRental feature) and also the ExpiredContract role (from the ReturnVideo feature), in order to be able to expire a contract for this Customer.

- Dependency between a role and a composition of multiple actors. This dependency is a dependency between a role and a composition of roles and a base-component. An example of this dependency is in the Return role of the ReturnVideo feature. The Video returned should contain the ContractMaint enance role and the RentingMaintenance role to be able to determine whether the Video is already rented and to add a new Contract to the Video if this isn’t the case.

The prototype implements the two role related dependencies with the help of the traditional Java dependency model (i.e. the use of the import statement). The two other dependencies are only partial realized. The recursive way actors are composed, make the use of traditional Java dependencies between a roles and an actors semantically different.

The second implementation issue is the instantiation problem. At the moment the roles are written it is not determined which class should be instantiated, because other features can be added/removed on the fly. Observe that the last actor of a component contains the complete composed behavior for that component; this is due to the recursive composition behavior of the actors. Each of the components has its own derived component, which should contain the complete composed behavior for that component depending on the selected features.

The derived component therefore could inherit from the last defined actor for the component. If during the derivation process the derived component confirms to this inheritance and has a stable name, then it can be instantiated in the different roles.

Another implementation issue is the traceability of roles, features, and actors, which is required for debugging the derived components. In the prototype the traceability of the different actors is accomplished by the first class rep-
presentation of the actors. The name of the package in which the actor class is defined is determined by the feature and role names. The name of the actor class is equal to the name of the base-component on which it is mapped, resulting in a complete reverse mapping from the derived components back to the roles, features, and base-components.

5.3 Summary

This section presented a prototype implementation of the feature model. The composition process used in the implementation was motivated and explained. The concrete form of features, roles and actors in the implementation was presented. Implementation issues as the lack of a dependency model in the feature model, traceability of roles, features and base-components, and an instantiation problem were discussed.

6 Related work

This section provides an overview of related work and their relationship with this paper. Separation of concerns, features, role modeling, and software product lines and software architecture are the four areas of interest related work is examined.

6.1 Separation of Concerns

Separation of concerns is the appliance of the divide and conquer paradigm on software design. By separating different concerns in separated entities the design becomes easier, but the “gluing” of the pieces becomes harder.

Subject Orientated Programming (SOP) [12][11] uses the concept of different views on an entity. Each view has its own object hierarchy. Composition rules define how the different object hierarchies can be combined into a single unified object hierarchy. Our approach differs in the focus, which is on the collaboration aspect of feature related variability and not on functional hierarchical differences.

Aspect Oriented Programming (AOP) [16] uses the concept of aspects to capture functionality that is cross cut in normal object decomposition. So called join points provide hooks to merge aspects with the objects. One of the implementations of AOP is AspectJ [17], here the join points are the method activations. A conceptual model stating what aspects are missing in AOP. The presented feature model in this paper can be used as a conceptual model for aspects, with the aspects implementing the composition of our feature model.

Multi-dimensional separation of concerns [25], as implemented in the HyperJ [19] approach, models different concerns in independent individual dimensions. Rules defining the relationships between the independent entities of the dimensions guide the necessary composition process for system generation. Our feature model can be viewed as a two dimensional instance of a multi-dimensional separation of concerns model. The first dimension is the concern of the base-components, the second the feature related variability dimension. The feature model is not restricted to only these two dimensions of separation of concerns, because no restrictions on the dimensionality of the base-components or features are defined.

The composition problem found in this paper (see Section 4) is universal for multi-dimensional separation of concerns, because each concern model will only describe a part of the behavior of an entity. However, each concern model needs to overlap/relate to other concern models, otherwise a composed view of an entity is not possible. Multi-dimensional separation of concern therefore has the inherent problem that an operation of an entity could have multiple behaviors that should be combined, resulting in a composition problem.

6.2 Features

Our approach is not the first one, which tries to model features in the solution domain. The Feature-Oriented Domain Analysis (FODA) [14] method is a method for identifying features during domain analysis. FODA uses the representation of feature trees to visualize the variability and dependencies of features. Later the Feature-Oriented Reuse Method (FORM) [15], which is a superset of FODA, was developed to prescribe how the FODA feature model could be used to develop domain architectures and components for reuse. The main difference between our approach and FORM is the traceability of features at the design and implementation level, which isn’t the fact with FORM. This traceability is lost during the FORM application-engineering phase.

Christian Prehofer [20][21], uses Feature Oriented Programming (FOP) to compose features into objects. FOP is an extension of the object oriented programming paradigm. It uses separate entities called lifters to model feature interaction and separates core functionality from feature functionality. Our approach differs in two ways; the first is that a first class entity (the actor) for the composed behavior is present, enabling the definition of a feature based on the composed behavior of two other features. The second difference is that a feature is not one static class but consists of different roles being mapped onto different domain components.

Pamela Zave [27] discusses a distributed feature composition technique (DFC) for telecommunication services. She uses a pipe & filter style architecture with the features being components, switches and routers connect the components with connectors to form a chain of components through which data can move. Components can be added and removed by the switch and thereby changing the current feature set. The main difference with our approach is the fact that we do not require a particular architectural style to be used and features do not have to be contained in a single component.

A more global view of how features can bridge the gap between problem and solution domain is presented in [26]. In their view features are composed out of requirements fragments and realized in one or more design fragments, which make up the complete design. This paper can be seen as a more detailed description of how the design fragments making up the feature can be modeled and composed for making the complete design.

The relationship between features and SPLs is also mentioned in [8], which proposes a feature-driven aspect-oriented product-line CBSE. The main idea is to use a fea-
ture-driven analysis and design method like FeatuRSEB [9] to develop a feature model. One or more aspect orientated implementation techniques [18] then can be used to implement features in separate code fragments, which in turn can be composed based on the selected features for a given composition. The global idea is the same as the approach in this paper, however the focus of this paper is more on the composition of features, whereas [8] is more a global modeling view.

6.3 Role modeling

The idea of role modeling is not a new idea. The OOram method [22] uses role models based on collaborations of different roles. Two or more role models can be synthesized to form a new and more complex role model. The general role modeling idea’s presented in OOram are used at various abstraction levels. The main difference with our approach is in the synthesis or composition of two role models. In OOram this is only done at a structural level that is only the structural requirements are validated, compositional problems have to be solved by the developer himself.

Role models can also be integrated into object-oriented frameworks [23]. The main focus of the role model used in this approach is on the interface part. A coupling between an interface and an implementation of a role is not made, something that our feature model does. The composition problem we have identified is therefore not relevant in there approach, because the composition problem finds its roots in a coupling between an interface and an implementation.

Fowler [6] defines design patterns that can be used to implement roles in an object orientated language. The main concept of the patterns is that the objects are dealing with a single object that has multiple changeable types. An object is therefore aware of the multiple typing of the other objects and has to act on this, which in the feature model does not have to be the case, because we want to be able to develop unrelated features independently.

The use of mixins [24], abstract subclasses representing a mechanism for specifying classes that will eventually inherit form a super class, is another approach to compose collaborations. The difference with our approach lies in the mapping of the roles of the collaborations to the objects. These can only be mapped to a single domain object and multiple roles cannot be mapped to the same domain object, something our feature model can.

6.4 SPLs and Software Architecture

Software Product Lines (SPL) [3] is an approach to develop software not on an application base, but on the basis of family of related applications. The commonalities between the individual products (applications) can be used to create so called common assets, which are reusable components that can be customized for the individual products. The field of variability management [10] of SPLs is mainly concerned how the differences between the products can be managed. This paper presents a method how the common assets can be customized for a specific product based on a selection of features and therefore is a form of variability management.

In the context of SPL [5] also use features to customize the common assets to derive a product based on a selection of features. They use packages as features and the merging of source trees to accomplish feature composition. The merging of the source trees takes place at so called variation points. A variation point in there approach is a simple switch statement, defining the difference variations on the code level. A problem with this approach is the definition of the variation points. The variation points have to be programmed out manually in the form of switches and this mixes the feature related code with the common asset code. This reduces the traceability and the reuse capabilities of the feature related code, because they don’t have a first class representation, which our approach has.

The Software Architecture (SA) [1] of each derived product is a variant of the SPL architecture. The feature model incorporates the components of the SPL architecture through the use of the base-components. The feature model can therefore modify the architecture of a product by adapting the existing base-components by mapping new roles of features on it and introducing new connectors and components resulting from features.

7 Conclusions and future work

7.1 Conclusion

In this paper we have investigated the potential of using features in the solution domain of Software Product Lines. Our main focus was on the design and implementation level. Starting at the design level a feature model was presented, modeling features. The model showed how features could be modeled as a collection of roles. The roles in turn can be played by different base-components resulting in actors. At the implementation level a way to implement the model is outlined. The model and the outlined implementation strategy is illustrated and validated with a prototype implementation.

With the help of a formalized version of the model a compositional problem is identified. The composition of two roles in an actor becomes problematic when both roles have different implementations for the same method. To solve the composition problem there are three potential solutions: skipping, concatenation and mixing. In the skipping solution only one implementation is executed, the others are skipped. With the concatenation solution the order in which the implementations are to be executed is defined. The mixing solution opens up the black box of the implementation and let developers specify specific composition points and behaviors. One or more of the three solution forms can be used to solve composition problems.

By predefining how the composition is done and which composition solution to use in the case of a composition problem, we can keep the composition of our feature model complete, consistent, deterministic and implementable. Making automatic product derivation in a Software Product Line based on a selection of features feasible.

7.2 Future Work

This section provides an overview of the remaining issues,
which form the basis for further work. The main remaining open issues are:

- **Automatic composition support.** An open issue of the prototype is the absence of a compiler that supports the composition process. At the moment the necessary composition steps still have to be done manually, which makes the application of the composition process a very time-consuming and error-prone one. However, we already have defined an algorithm for the composition process, which can easily be programmed out, automating the composition process.

- **Scalability of the feature model.** Scalability of the feature composition model is one of the main aspects that demand additional validation. Although the feature composition model was designed for the use in Software Product Lines, it has not yet been demonstrated if the model can be scaled up to this level of scale.

- **Lack of a dependency model.** The feature model does not include a dependency model. The combinations of features that are possible for product derivation are directly related to the feature and roles dependencies. So, it is important to extend the feature model with a dependency model. The first steps have already been taken in Section 5.2 where four dependency relation types are already identified.

- **Validation of the feature model.** A correct and complete validation of the feature model is difficult to accomplish. Cases can be used to validate the feature model, for example the video-shop case in this papers does.

These open issues form the basis for further work. We would like to investigate how automatic composition support can help with the scalability of the feature model. For decent automatic composition support the feature model should be extended to include a dependency model. Additional research is planned with an additional case helping us to investigate the scalability of the feature model and providing additional validation of our approach.

8 References


