A component model for applications based on feature models

Detlef Streitferdt, Ilka Philippow, Christian Heller
Ilmenau Technical University
{detlef.streitferdt | ilka.philippow}@tu-ilmenau.de

Abstract

Current software development requires very short release times and at the same time high product quality. Prefabricated components used within a system family development fulfill these needs. This planned reuse has to be considered in all phases of the system family development. Feature models describe the system family in an early stage of the development cycle, whereas components are used to describe the structure of an application at implementation time. We are facing a gap between the feature model as starting point to process the requirements of the customer and the component model used to describe and derive an application as member of the system family.

In this paper we propose a component model for the development of system families. This model is integrated into a feature-driven development. The integration of both models and the definition of the component model allow the automated derivation of system family members.

Keywords: Family component model, feature model.

1 Introduction

Modern software products shall be developed within a short time and at the same time they should be of a high quality. Software engineering is able to fulfill these requirements by prefabricating components. Within a domain, planned and comprehensive reuse of components is supported by the concept of system family development. A system family is based on a reference architecture made of assets, which are common to all family members and assets, which are variable. Commonalities and variabilities have to be considered in all phases of the system family development.

With features, introduced in [1], commonalities and variabilities of a system family can be modeled. Based on a selection of features an application as member of the family can be derived. This application, composed out of the prefabricated components, has to meet the customer requirements defined for the features, the interaction of features and the corresponding components. The feature model is the basis for the component architecture of the system family. Thus, a component model is needed to reflect the above mentioned requirements of the feature model.

In this paper we propose a component model for the automated derivation of family members based on feature selections. The key issue of the proposed model is the relation of features to components and between components, which leads to a first version of the system family architecture and enables the automated creation of family members.

2 State-of-the-Art

A way to describe and model variabilities of system families by means of features was initially described by [1] in 1990 and further developed in [2], [3] and [4]. Features describe the system family for a future user, so that he can choose an own application, based on the features of the family. Features should describe an outstanding, distinguishable, user visible aspect, quality or characteristic of a software system or system. Based on own experience and analyses made by [2] and [4], features are very well suited for users or customers, to understand the system quickly and thus, make a sound choice of their desired system based on the features offered.

Features are arranged as a tree, the root of the tree is the concept node, representing the system family itself. Every feature can be optional or mandatory. Feature leaves marked mandatory back over all levels of the tree up to the root node, form the core of the family. In contrast to this all other features model the variabilities of the family.

As shown in Figure 1, features are hierarchically organized starting with the concept node at the root of

the tree. As shown, all feature leaves with only mandatory markers up to the root of the tree belong to the core of the system family, in the picture these features are *MPEG-2*, *Video* and *Case*,. The other features form the variabilities of the family, for example one could choose *DivX* playing capabilities or not. With the requires and excludes constraints we can further limit the possible choices of features in the tree. For a given feature we can state, that another feature is required or must be excluded, as modeled for the *Parental Control* feature, which is based on locking the remote control and thus useless with the possibility of a direct control panel at the hardware system. A valid selection of features represents a desired application as member of the system family.

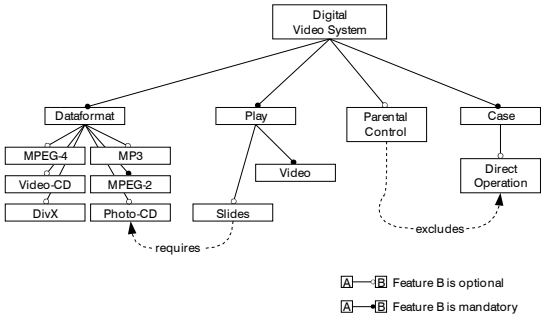

Figure 1: Feature Model of a Video System

Extended Feature Models

In [5] feature models are extended by constraints, that can be processed in an automated way. Over 20 new constraints are pre-defined and available for a feature developer. As shown in Figure 2 constraints can be defined between several features. Here a mathematical constraint "m" is included. In our video system example, we process digital TV signals, referred to by the *DVBCard* feature.

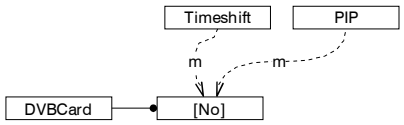

Figure 2: Extended Feature Models

As mandatory parameter feature the number of DVB cards is required. As soon as the *Timeshift* feature is chosen, the first constraint "m" has to be true. This constraint defines, that the number of DVB cards has to be greater than two. The same constraint was defined for the *Picture In Picture* (PIP) feature, since at least two DVB cards are needed to watch two different channels at the same time.

All constraints are defined in a language similar to the Object Constraint Language (OCL) [6]. Thus, the feature model can be checked in an automated way for consistency and each feature set can be validated against the constraint rules. As a result, we can proceed the development with valid feature selections, which are taken as input for the component model described in this paper.

Component-based Applications

Components encapsulate and provide functionality on an abstract level through interfaces. The functionality is not bound arbitrary, it is more an encapsulation of functionality of autonomous concepts or processes out of the same domain. This form of encapsulation gathers knowledge out of a specific domain [7]. Current components definitions for component based development, like [8], [9], [10], [11] and [12] specify components and their relations to other system parts. For the proposed component model the interface of components is important and has to be well defined. Thus, we extended the component definition of [8] by the definition of interfaces described in [13]. A component interface is now divided into *provided* and *required* parts.

In this paper we propose a new component model based on feature selections. Using this component model we are able to automatically derive applications as members of a system family. The novelty of our model is the integration of components and features to support the automated application creation. We relate features to components and are able to directly derive an early version of the system family architecture.

3 Features towards a component model

The first step towards our component model is the definition of five features types.

1. *Functional features*, based on [5].
2. *Interface features*, based on [13].
3. *Parameter features*, originally defined in [4].
4. *Structural features*, as defined in [5]
5. *Conditional features*, newly introduced in this approach.

For our component model we assume, that a customer will choose system based on the desired *functional features*. Each component of the system family will have at least one functional feature. Components without functional features won't exist. Parameters are attached to configure functionality and are assigned to functional features. As an example consider the re-

cording capacity of a digital video system. The functional feature "record video" should have a capacity parameter with a value for the hours of recording time.

As the next type we have *interface features*, describing a component with its required and provided interfaces. In Figure 3 an audio conversion component with four interface features is shown.

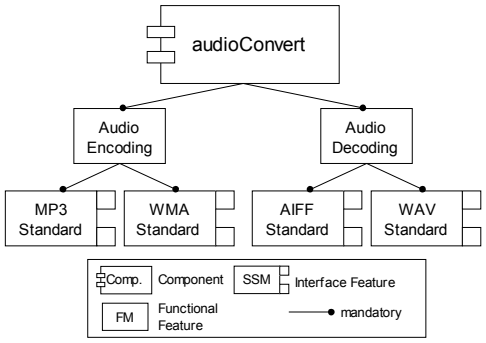

Figure 3: Audio Conversion Component

We know which interfaces the component is going to support. But we don't know whether the component is able to convert each of the decoding formats into an encoding format. Thus, we need to extend the component model by functional features, as described above. The audio conversion component has three new functional features, as shown in Figure 4.

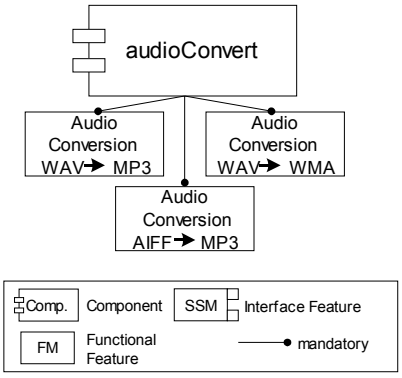

Figure 4: New Audio Conversion Component

Now the interfaces and the possible conversions are completely modeled. For the initial derivation of an application out of the system family interface features are only of minor interest. Customers are mainly interested in the functional features they can buy. The importance of interface features rises when it comes to an update of existing systems. Here new components can only be integrated if all interfaces are known. The graphical notation for interface features is depicted in Figure 4.

Components don't have *parameter features*. As stated above, parameter features are used to configure functional features. Thus, they directly influence the configuration of a component. Parameter features are typed values.

Conditional features are relevant for an automated choice of components. They represent, just as parameter features, typed values, like the size of a hard disk or the price of a feature. They can also represent weak features, like the license model for a component, which will not be numerically expressable. Functional features may also be further described by conditional features. They refer to features not valid for the whole component, but only for the functionality in question, like the sound quality or compression rate of videos

Conditional features can additionally support the decision process. For conditional features, as typed values, an automated decision process is possible, but for weak features only a manual decision process is practicable. Taking the license model example above, most developers might understand the differences between an *Open Source* license and the GNU license. Legacy license models are mostly hard to understand – in bigger companies they will be checked by legal departments. For such features only a manual decision process is applicable. We don't address these features in our model, this is left for future research projects.

Structural features don't influence the choice of components. They only summarize their sub-features and improve the readability and the overview of feature models. If a functional features is decomposed by a set of new functional features, in most cases, the original functional feature will become a structural feature. For the architectural development of the system family structural features are a good hint for components to be build. Structural features can describe component and all functional features through their sub-features.

For an automated choice of components we make use of the following four feature types, where *structural features* are not used for choosing components. *Functional features* represent the primary requirements of a customer. The customers choice is mainly influenced by functional aspects. *Interface features* determine, which components will be part of a possible choice. The future application must have an interface for the integration of further components. *Parameter features* describe configurations of adjustable values for functional features of a component. *Conditional features* are constraints used for the choice of compo-

nents. They can be automatically or manually evaluated and are used to assess and narrow the component choice.

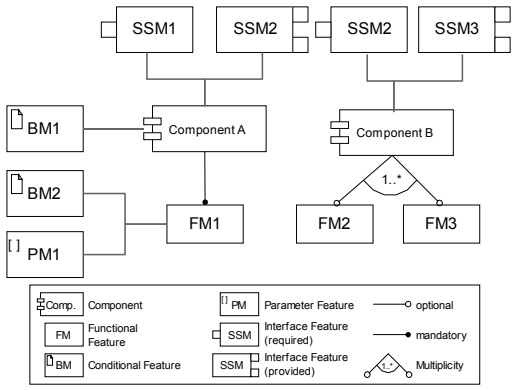

Figure 5: Overview of feature-component relations

All the possible relation between features and components are summarized in Figure 5.

Working with components

The assembly of components [8] is based on the constraints and requirements elaborated in the feature and component model and is performed by a *component assembler*, a new role within the system family life cycle. Features of components, that are not part of the feature model can not be considered for application of the system family. The features of the feature model must have their counterpart in one of the components. Optional features have to be optional in a component, mandatory features can be optional in a component – an always enabled optional feature of a component is externally seen as mandatory. For components implementing only one feature the whole component may be part of an application, to realize the conditions of the feature model. All the possible combinations of optional and mandatory features in the feature and component model are summarized in Table 1.

Feature Model	**Component**	
optional	optional	allowed
mandatory	optional	allowed
optional	mandatory	Allowed only if component can be left out of the application, without influencing other components.
mandatory	mandatory	allowed

Table 1: Possible combinations for the realization of features

To avoid implementation problems it is advisable to either implement a single feature per component or implement more feature in a single component, where each feature is optional – can be "switched off".

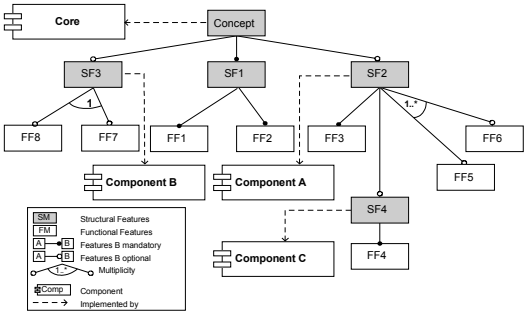

Figure 6: Example Application

As shown in Figure 6, the architecture of the system family is strongly influenced by the feature model, by the relations between features and the combination rules. The core of the family is depicted by the functional features FF1 and FF2. In addition, the core of this simple example is build out of one component, referenced by the structural feature SF1. Thus, structural features are a hint to components. In this example the features are modeled in a way, that components don't have overlapping features, what leads to a one to one relation between features and components.

As an example for a more complex interrelation between components a sound studio system family was prototypically developed. The sound studio has an open architecture, so everyone is able to develop extensions for this system. New functionality can be developed by integrating new components into the system family. In Figure 7 this example is shown, the core of the family is not included in detail, just a link to the core component is present.

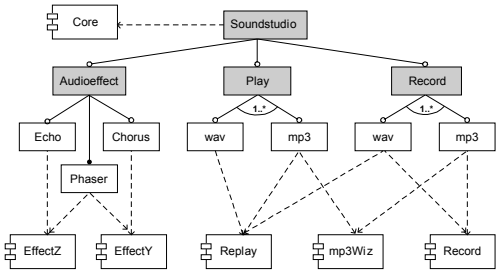

Figure 7: Sound Studio example

For the sound studio the structural features *Audioeffect*, *Play* and *Record* are optional. As shown, a customer would have the choice between several data formats for playing or recording and several audio effects to change sound streams. Every functional feature has a relation to one or more compo-

nents at the bottom of Figure 7. As stated above, components should offer their functionality in an optional way, to overcome the problems addressed in Table 1. The problem with the decision process in this example is the relation of structural features to components. Which component should finally be taken to fulfill the requirements of a given feature? Without additional decision support we would end up with a choice by chance.

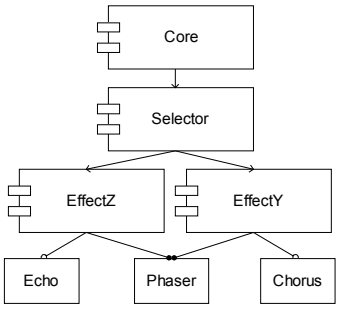

Figure 8: Selector component

Our component model supports decisions by conditional features. The preferences of a customer are mapped to the conditional features. In case the customer has not enough knowledge of the domain the component assembler will have to guide the customer. We distinguish between special conditional features, referring to single functions in the component and general conditional features, which can be valid for all components. For the sound studio example in Figure 7 a special conditional feature would be the recording quality *Record – MP3*.

Another issue is the usability of conditional features. The license model is without doubt more complicated than the price. Here we found, that either the preferences of the customer have to be clearly related to conditional features or they have to be clearly distinguishable (for example smaller than or greater than).

If a customer wants to have the *Echo* audio effect and the *Chorus* effect at the same time we run into a problem with the *Phaser* feature. The given customer requirements lead to components *EffectZ* and *EffectY*. Since Phaser is available in both components, which *Phaser* implementation should be taken? For obvious reasons, both are not possible. To solve this problem we introduce a *Selector* component. It stores configuration information about the *Phaser* implementation, that should be taken. The *Selector* component, depicted in Figure 8, can be a simple dialog asking the user to select the desired implementation or it can be a passive component using a configuration file, containing the information which implementation to take based on the domain knowledge. In case a *Selector* component is not a possible solution, we can not meet the customer requirements.

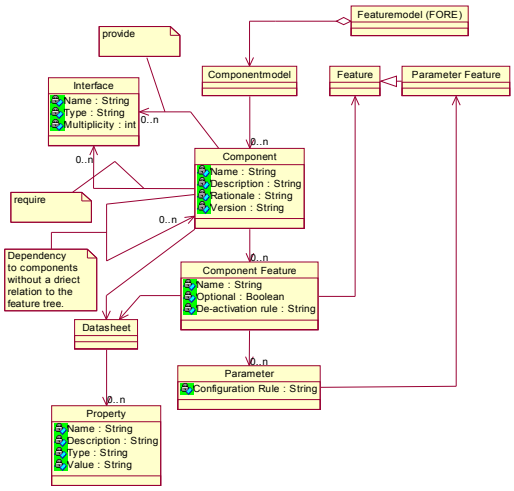

Figure 9: Proposed component model

Component model

The component model is shown in Figure 9. A *Datasheet* is attached to each component, to support the automated selection of conditional features of a component. The *Datasheet* holds a list of *Properties* for a more detailed description of the component, like the price, the producing company or the license model. A *Component* knows about its *Component Features* which provide information for the derivation of an application of the system family. *Parameters* for a specific *Component Feature* provide a *Configuration Rule*, which describes the programatic steps to be performed to use the *Parameter*. For *optional Component Features* a *De-activation* Rule describes the programatic steps to be performed to disable this feature. The *Component Feature* and the *Parameter* have references to the feature model. *Components* hold two lists for their *Interfaces*, the one for the *required* interfaces, the other for the Interfaces the components *provides*. With this information only valid component choices are possible. Providing and requiring interfaces must match. The maximum number of simultaneously usable interfaces by a component is expressed with the multiplicity. If more than one component is using an interface, we assume the component providing the interface has a management function for the parallel access of its functionality. In addition, each component holds a list of required components, which are not part of the system family.

We haven't defined a specific language for the rules in the model. The language is dependent on the software system, which is used to analyze and execute the model. Our prototype uses an Ant-file [14] for each component. This file contains processes to toggle and change component parameters, described in the Ant scripting language. Our system for deriving an application calls the Ant-scripts for the desired configuration. A rule to switch off a feature in a component simply calls an Ant-target, for example “disableFeature”. By using a script language like Ant we can easily use different components, since access and usage of component features differs largely.

4 Conclusion and further work

In this paper we propose a component model for the development of system families. This model is based on a feature model, which describes the family itself and all the relations between the features. This feature model can be analyzed and automatically checked for consistency and feature choices can be checked for validity. The feature types of the feature model are extended by this paper. Thus the feature model structure directly reflects a first version of the component architecture of the system family.

The component model holds all information for an automated derivation of system family members, specified by a set of feature and selected based on the customer requirements. For components all the information needed for “adjusting” the component is stored in the component model.

For a complete development method the development process, currently existing for the development of the feature model, has to be extended to support modeling components as well. We are currently working on the development process for the professional usage of the component model. For the future we plan to integrate the development processes of the feature and the component model, for a more elaborate system family development process. In addition we plan to extend the prototypically existing tools, like the described Ant-scripts, towards a complete tool chain for the whole development process.

5 References

[1] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, A. Spencer Peterson, "Feature-Oriented Design Analysis (FODA) Feasibility Study", Report: CMU/SEI-90-TR-21 ESD-90-TR-222, www.sei.cmu.edu, 1990.

[2] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim,Euiseob Shin, Moonhang Huh, "FORM: A Feature-Oriented Reuse Method", Pohang University of Science and Technology, www.postech.ac.kr/e/, 1998.

[3] Kyo C. Kang, Kwanwoo Lee, Jaejoon Lee, "Feature-Oriented Product Line Software Engineering: Principles and Guidelines", Pohang University of Science and Technology, www.postech.ac.kr/e/, 2002.

[4] Krzysztof Czarnecki, Ulrich Eisenecker, "Generative Programming: Methods, Tools, and Applications", Addison-Wesley, 2000.

[5] Detlef Streitferdt, "Feature-Oriented Requirements Engineering", Dissertation at TU-Ilmenau, 2004.

[6] OMG, "Unified Modeling Language Specification", 1999, www.omg.org.

[7] Andresen Andreas, "Komponentenbasierte Softwareentwicklung mit MDA, UML und XML", Carl Hanser Verlag, 2003.

[8] Szyperski Clemens, Gruntz Dominik, Murer Stephan, "Components Software - Beyond Object Oriented Programming", ACM Press, 2002.

[9] Colin Atkinson, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqua, Dirk Muthig, Barbara Paech, Jürgen Wüst, Jörg Zettel, "Component-Based Product Line Engineering with UML", Addison-Wesley, 2002.

[10] Andreas Hein, John MacGregor, Steffen Thiel, "Configuring Software Product Line Features", Springer Verlag, 2001.

[11] Schryen Guido, "Komponentenbasierte Softwareentwicklung in Unternehmen", Deutscher Universitäts-Verlag, 2001.

[12] Wallnau Kurt C., Hissam Scott A, Seacord Robert C., "Building System from Commercial Components", Addison-Wesley, 2002.

[13] Bosch Jan, "Design and Use of Software Architectures, Adopting and evolving a product-line approach.", Addison-Wesley, 2000.

[14] The Apache Ant Project, "Ant - A Java Build Tool", 2004, ant.apache.org.