Composition of applications based on software product lines using architecture fragments and component sets

Marco Körner, Sebastian Herold, Andreas Rausch
Clausthal University of Technology
Department of Informatics
Julius-Albert-Str. 4
38678 Clausthal-Zellerfeld, Germany
{marco.koerner, sebastian.herold, andreas.rausch}@tu-clausthal.de

ABSTRACT
Software product lines (SPL) are a popular concept to represent commonalities and variability among a family of software systems. In many approaches to SPL engineering, feature modelling is used to specify which common and different features instances of an SPL can have. By mapping features to components realising that feature, the software architecture of the SPL instance can be derived from the features it requires.

However, many approaches to feature modelling and mapping to implementation ignore the fact that features are often not implemented by a single component alone but by a set of components combined in a specific way. Moreover, they are often unable to capture implementation alternatives for features.

In this paper, we motivate the need for a more extensive way of mapping features to alternatives of architecture fragments by an illustrative example. We discuss the challenges of composing a software architecture out of architecture fragments and outline a solution approach.

Keywords
software product line, component based development, software architecture

1. INTRODUCTION
A software product line (SPL) is a set of software systems sharing a common set of features and that are developed from a common set of core assets. By capturing the common assets of the software systems and managing the differences/variability in a controlled way, SPL is said to make software development more efficient by systematic reuse [1].

The main parts of SPL engineering are domain engineering and application engineering. During domain engineering, the common and variable features of software systems of an SPL are captured in a feature model, a reference architecture is defined, and a repository of components for constructing software systems is maintained; exemplary approaches are FODA [7] or FORM [8]. In application engineering the application is instantiated by selecting a set of components that implement the selected features and connect them according to the reference architecture [10].

To be able to instantiate a software system during application engineering, a mapping between the actual features required by the software system and the available components must exist. [13] mentions that several methods based on FODA use a one-to-one mapping between features and single components. But these mappings do not allow for the following possibilities: A feature may be implemented by a complex structure of components which are connected and interact in a specific way or there may be alternative feature implementations.

Approaches using an n-to-m mapping allow implementing complex component structures and implementation alternatives. However, these alternatives cannot be modelled independently. Consequently, we introduce coarse grained architectural fragments describing the required structures of alternative feature implementations independent of each other.

In this paper we discuss how variability on an abstract architectural level can be addressed by modelling and composing architecture fragments. It deals with the question of how a domain model could be defined to meet the needs that arise during the application engineering process on that level.

The following section gives an example motivating the need of expressing variability in terms of architecture fragments. Section 3 describes the work that has already been done in this context and existing approaches dealing with the problem of a high diversity on an architectural level. Section 4 presents a new approach to generating an application concerning the variability within software architecture and used component sets. This paper ends with a conclusion which includes a small glimpse onto future work.

2. EXAMPLE
In embedded systems, structures similar to the one shown in Figure 1 can often be found. A sensor component delivers data to a controller which then sends data to an actuator component. One variation point is defined by the fact that different types of sensors can be supported.

To support graceful degradation, some applications can use a second sensor if the first one fails. Therefore, some applications have to be able to process only one sensor sig-
Figure 1: Structure of a simple control application

Figure 2: Structure of a controller application with multiplexed sensor data

A similar structure can be built with two alternative feedback controllers connected by a special demultiplexer component. The diagram in Figure 3 represents the feature model of an example in a FODA-like syntax [7]. Controller A and controller B are implemented by a specialised set of components consisting of a feedback controller and an actuator-component each.

Figure 3: Example feature diagram

It can be seen that the implementation of a feature (e.g. the ability to process data from sensor A) consists of a complex structure of components but not a single one. However, the complex structure could be understood as a single component encapsulating its “inner” parts but such a component would contradict the understanding of components as “independent units of deployment” (in contrast to the single components) [12]. Hence, a mapping of features to single components is insufficient.

In addition, alternative implementations should be developed by different teams. For this we need the ability to model these alternatives independently.

3. RELATED WORK

Several approaches improve the possibility of modelling solution space variability using different techniques compared to FORM-based methods. These techniques can be classified as negative and positive variability [14]. Negative variability is represented by annotative approaches and superimposed variants. One monolithic model represents all products of the SPL. Special annotations to model elements define whether the element will be used at build-time of the application or not. Positive variability is realised by a core system and a set of new parts that can be added to the core in the application engineering process.

One example for superimposed variants is given by Czarnecki and Antkiewicz with their template approach [2]. The presence conditions allow an n-to-m mapping between features and components or connections respectively. Presence conditions are modelled as annotations on model elements. Every alternative feature implementation is part of the same model. This allows the reuse of components for the implementations of different features. But using this method it is difficult to manage large-scale systems with a high amount of diversity [11].

Haber et al. give an example for positive variability. “Delta modeling is a transformational approach for modular variability modeling” [6]. A core architecture that already describes one application of the SPL can be used to generate the architecture of every other product by applying architectural deltas. These deltas describe the differences of an application to the core concerning the use of components and their interconnections. It is possible to implement an n-to-m mapping but this could result in conflicts between deltas that affect the same architectural elements [5]. To resolve these conflicts, constraints for the order of delta application have to be specified.

Another interesting approach is pattern-based [4], developed by Street Fant et al. Pattern specific features can be mapped to architectural design patterns. Variability within those pattern-based structures is influenced by pattern variability features. The idea of mapping features to coarse-grained structures of the system can be very helpful in large-scale systems. But the need for special pattern-specific features increases the complexity of the feature model. On the other hand, the one-to-one mapping between features and patterns or components respectively reduces flexibility, since alternative implementations of a feature are not possible.

The presented examples do not allow implementing alternative features independently using coarse-grained architectural structures.

4. ARCHITECTURE COMPOSITION

The goal is to combine the advantages of the presented works, which are:

- modular solution space variability modelling,
- mapping between features and components as well as coarse-grained architectural fragments and
- alternative feature implementations.

The basic idea is to assemble the application’s architecture using overlapping architectural fragments. In this section a new approach to generate an architecture and our proposal of a modified domain model will be described.

4.1 Domain Model

Although the previously mentioned domain models do not fulfill the needs, some of the basic ideas can be reused for the new approach. The new domain model consists of a feature model and a component repository, but instead of using a basic architecture, a repository of architectural fragments will be defined. To compose an application using these domain model artefacts a mapping between them is provided.

4.1.1 Feature Model and Component Repository

Since there are no special requirements for the meta models of the used feature model and components, a FORM-like syntax [8] describes the feature model and UML component diagrams present the components [9].
Figure 4: Artefacts of the new domain model and their mapping associations

Figure 3 presented what the feature model of the example looks like. For the component repository in our example, the following components are needed: two different sensor components, two controller components, two actuator components, a multiplexer and a demultiplexer.

### 4.1.2 Architecture Fragments

Architecture fragments describe parts of the application. These can be used to define how a set of components must be connected to implement the corresponding feature. Thus, it is possible to describe the structure of reusable patterns or how components can be integrated into an environment.

Fragments consist of parts and connectors. Parts represent roles that a component can take within the fragment. Roles define the preconditions components must satisfy to be that part. That includes for example the association to a component that takes another special role.

Figure 5: Collaboration diagrams represent fragment examples

Connectors describe the required associations between the parts or components, respectively. That means that there may be more associations between components but if they are not needed to implement a special feature, they will not be modelled as connectors within the corresponding fragment.

UML collaboration diagrams can be used to describe fragments. Figure 5 shows the fragments that are used in the example. A component that can take the role sender in the fragment has association to a component that can take the role receiver.

### 4.1.3 Mapping between the Artefacts

To derive an architecture of an application it is necessary to define a mapping between features, fragments and components. A feature can be connected to several fragments and every fragment can be connected to several features. This defines an n-to-m-mapping between features and fragments. Figure 6 shows that the feature sensor B - processing can be implemented by using either multi-sensor-fragment or sender-receiver-fragment.

On the other hand, features can be connected to several sets of components that can be used in the implementation. And every component can be part of the implementation of several features. In the example, the feature sensor B - processing is connected to sensor B (see Figure 6) and the set consisting of sensor B and multiplexer.

To know how the components have to be connected, components are placed into the parts of the fragments. This can only be done if it is known which component fits into which part. Therefore a mapping between the roles of the parts with the components that can play these roles is defined. In Figure 6 the role of sender can be taken by the components sensor B and feedback controller B.

Using all three kinds of association it is possible to say that the feature sensor B - processing in the example can be implemented by using the component sensor B as sender in fragment sender-receiver-fragment.

### 4.2 Application Engineering

The goal of the application engineering process is to instantiate one product of the SPL. This means that fragments and components have to be combined into the architecture of the configured application.

The idea is to connect different fragments using components, just as two metal plates are fastened together by rivets. The fragments are overlaid so that parts of different fragments are filled by the same component if it can fit all corresponding roles.

In the example the feature sensor B - processing can be implemented by using the fragment sender-receiver-fragment with the component sensor B as sender. One implementation of controller B consists of the same fragment but another set of components. In this implementation the component actuator B is the receiver and feedback controller B the sender. The latter component can also take the role of receiver in the fragment implementing feature sensor B - processing. This characteristic allows one instance of that
component to take both roles, which can be used to connect the involved fragments (see Figure 7).

**Figure 7: Two fragments are connected to a new one**

This can cause a problem if it is possible to instantiate the potential rivet-component more than once, because there could be two unconnected fragments instead of a combined one, which results in an invalid architecture. The example in Figure 7 shows that a component can take more than one role of the same fragment. Feedback controller B can be used as sender and as receiver. It means that there would be more than one solution to fill up parts with components. Every possible solution must be considered.

To handle potential prohibitive constraints must be defined, which must be followed to validate the application. These constraints specify, for example, the context in which a component can be used to connect fragments. Universal rules still need to be defined. Due to the restrictions of the component model used in the domain of the example [3] the only point left to consider was not having any unconnected fragments.

The first step is to configure the application by choosing the features the application should provide using the given feature model. The mappings of the domain model are used to derive sets of fragments and sets of components from the set of features. Every possible feature implementation must be generated before the architecture fragments are connected to an architecture of an application in accordance to the defined rules. If more than one architecture can be generated, the architect must decide which solution should be used to instantiate the software system.

5. CONCLUSION

Merging of architectural fragments and components can be used to derive products of a software product line. It is possible to define an n-to-m-mapping between alternative coarse-grained structures and features as well as an n-to-m mapping between component sets and features. The possibility of modelling different feature implementations independently also provides an advantage for the evolution of the SPL. Adding new features in the SPL requires a new mapping, but not a change of existing mappings or models respectively.

The next step concerning the presented approach is to formalise it. An efficient algorithm must be defined to allow an automatic architecture derivation from a set of features.

In the future this approach will be extended by the possibility of using more than one instance of each component for the same application. Therefore, universal rules need to be defined, to identify fragments that can be connected using rivet-components.

6. REFERENCES