Mapping Feature Models onto Component Models to Build Dynamic Software Product Lines

Pablo Trinidad, Antonio Ruiz–Cortés, Joaquín Peña and David Benavides
Universidad de Sevilla
41012, Seville (Spain)
{ptrinidad,aruiz,joaquinp,benavides}@us.es

Abstract

Systems such as adaptative and context–aware ones must adapt themselves to changing requirements at runtime. Modeling and implementing this kind of systems is a difficult operation. Software Product Lines (SPL) approach has already coped with modeling a set of software products that share a common base of features by means of feature models. We propose using feature models to model the potential states of a product in what it is called Dynamic SPL. The objective of this paper is generating a component architecture that supports the dynamics of products and which is easily inferred from a feature model. The resultant model performs an automated analysis of the feature model in real–time to correctly response to changes.

1 Introduction

Context–aware and adaptative systems that must reconfigure to adapt themselves to changes in requirements or context, are more and more common. Fields such as ambient intelligence where systems must response to changes in the context and real–time systems where changes in requirements cannot harm the system availability are demanding this kind of dynamic systems.

Software Product Lines (SPL) intend to build a set of products that share an important amount of features paying attention to product commonalities.

SPL may be the approach that fits better to build dynamically adaptable products because modeling techniques that represent an SPL can be used to describe all the products which can derive from the original one. Feature modeling is a modeling technique proposed in the FODA method [10] to describe all the products in an SPL in terms of their features. A feature is a distinctive characteristic of a product that may refer to a requirement[9], a component in an architecture[7] or pieces of code[3, 8]. It seems intuitive to use feature models to model the potential states or configurations of a product

A main problem is building an architecture that dynamically adapts itself to changing requirements. In this paper we propose a process for the generation of a component architecture from a feature model. The generated architecture is able to activate or deactivate features making use of a configurator component that performs some analysis operations on feature models to make decisions.

This paper is structured as follows: Section 2 summarizes the current proposals to produce a software architecture from a feature model. Our proposal to transform feature models into component models to implement a dynamic SPL is presented in Section 3. A case study that applies our ideas to a TV production and broadcasting system is presented in Section 4. Section 5 remarks the feature model analysis operations needed to support the dynamic behaviour of the SPL. Lastly, we give some conclusions emphasizing the extension points of our proposal that will guide our future work in Section 6.

2 Mapping feature models to software architecture

The aim of inferring a design or an implementation from feature models is the common point of several contributions in software product lines. we focus on those that propose a direct mapping from feature models onto component models. Among all these works we distinguish three that can summarize the state of the art:

- Liu *et al.* [12] proposes a natural mapping from a feature model onto a software architecture. They consider a feature to be a *higher–level abstraction of a set of relevant detailed software requirements, and is perceivable by users or customers*. So a feature model is a requirements model whose functional and non–functional requirements can be mapped onto a software architecture. Based on this assumptions, they propose a feature–oriented requirements modeling process to build a feature model with the software architecture in mind. Although this work is very promising in its first steps, only some traces and guidelines are given to manually producing a software architecture from a feature model.

- Sochos *et al.* [14] proposes the Feature–Architecture Mapping (FArM) as a way to progressively transform a feature model into architectural component model. They establish the precondition that the feature model has been designed by thinking in components. Four transformations are proposed to sequentially producing a software architecture. It is worth to mention the special emphasis on categorizing the types of interact and hierarchy relations which helps on defining the relations among components based on the relationships among features in the feature model. Although it is an interesting proposal, they lack of a domain analysis method to produce the initial feature model and no support is given for a dynamic SPL. First issue can be solved by using existing domain analysis methods such as FODA [10] or [12]. Dynamic SPL support is slightly commented by suggesting the usage of design patters to support runtime variability but it is not a main issue in their method as it should be in dynamic SPL.

- Feature–Oriented Reuse Method(FORM) [11] relies on the FODA proposal [10] and describes how a feature model is used to develop the domain architecture and components for reuse. Guidelines and a process are given to develop these artifacts, but no support for dynamic SPL is given.

Analysing these proposals, they share the premise that a feature can be mapped onto a component so the feature model must be defined with this point in mind. Although a manual process is given to produce architectural models they do not place the emphasis on producing a dynamic SPL, i.e. no solution is given for dynamically changing a product by activating or deactivating its features at run–time.

With the aim of filling this gap, this paper focuses on the production of a component model which is built with the evolutionary and dynamic issues of a dynamic SPL in mind.

3 Our proposal

In order to produce a component model that represents the software architecture of a dynamic SPL, we assume that:

1. A feature can be mapped onto a component in the component model, and
2. The feature model is built thinking about component structure.

We propose following 4 steps to produce a traceable component model from a feature model:

Step 1: Defining the core architecture

The core of our component–based software architecture will be compounded by the features that are common to every product. First, we have to extract from the feature model which are those features. How this operation can be performed is described in next Section.

Once the core set of features is obtained, we start defining the component model by creating a component for each feature. The component responsibilities will be those assigned to its respective feature.

The components will connect among them depending on the relationships among features in the feature model. For each hierarchical relationship between a parent feature and a child feature, a dependency from the parent component to the child component is created. For each cross-tree constraint (*depends* and *excludes* relationships) a dependency in the direction of the constraint is created between the respective components.

Step 2: Defining the dynamic architecture

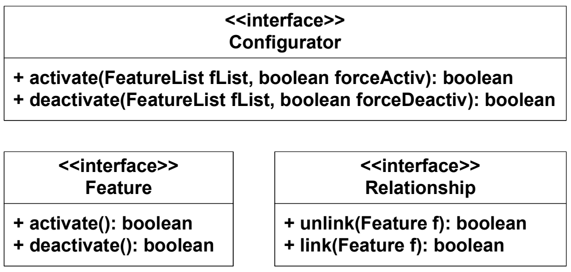
The non–core features are used to generate the dynamic architecture. Analogously to first step, a (feature) component is created for each non–core feature, and its responsibilities will also be those assigned to the feature. A *feature component* will provide a set of interfaces that will vary from its responsibilities, and will require some functionalities to its child features by means of input interfaces. It is not the objective of this paper to propose a method for extracting information about the provided and required interfaces of each component.

To connect the features each other, a (relationship) component will be created for each relationship in the feature model that connects at least one non–core feature. A relationship component joins those features that are connected by the respective relationship in the feature model. It provides and requires the interfaces of the feature components that it joins, acting as an intermediary among feature components and reducing the coupling among them.

As a last consideration, the relationship components that link a core and a non-core feature will be considered as a part of the core architecture.

Step 3: Adding the configurator

A configurator is the component in the architecture that provides the dynamic behaviour for a product. Among its responsibilities we enumerate the following:



<<interface>>
Configurator
+ activate(FeatureList fList, boolean forceActiv): boolean
+ deactivate(FeatureList fList, boolean forceDeactiv): boolean
<<interface>>
Feature
+ activate(): boolean
+ deactivate(): boolean
<<interface>>
Relationship
+ unlink(Feature f): boolean
+ link(Feature f): boolean

Figure 1. The interfaces described to provide the dynamic behaviour of a product

- It knows the feature model.
- It centralizes the feature de/activation requests.
- It checks for the requests to produce valid configurations.
- It delegates to the respective features the responsibility of de/activating themselves and their relationships.
- It decides which features must be de/activated as a consequence of another feature de/activation.

The configurator component provides an interface for the feature components to communicate it the de/activation of a set of non–core features. A description of the operations that the `Configurator` interface provides, are depicted in Figure 3.

The configurator will communicate with every feature for de/activation. This is the reason why every feature component must provide a `Feature` interface that allows the feature de/activation. The concrete interface operations are depicted in Figure 3.

A feature component is coupled with relationship components that must be aware of any de/activation that affects the features it links. For this reason, all the relationship components must provide a `Relationship` interface for the coupled features to communicate any change in their state. Communicating these changes is responsibility of the affected features and never of the configurator component.

Step 4: Defining the initial product

An initial product will be defined by a selection of the non–core features that will be initially active. The configurator component will be in charge of activating the selected features when the product is firstly launched. It is important to previously validate the configuration.

4 A Case Study

The mapping described in this paper has successfully been applied to generate an industrial real–time television SPL[6]. The objective was developing a system that could be adapted to changing requirements coming not only from a customer but many potential customers. An SPL approach was used to develop the system and a feature model used to describe the SPL requirements.

The system aims to broadcast a video composed by software, mixing TV signals, stored videos, Flash animations and any other kind of images or layers. Some kinds of effects wanted to be applied to the layers, such as black and white effect and lummakey and chromakey effects (remove colors ranging on a chromatic or luminance range). Different user interfaces (UI) are needed to interact with the application. At least a basic UI that allows managing layers and effects is required. Other UI are demanded to schedule TV compositions and to download SMS messages from a server and sending it to a Flash animation. From this information, the domain analysis process elicits a set of requirements for the products in the SPL:

- A product should draw an image as a result.
- Several layers compose the final image, such as TV signals, stored video and Flash animations. New layers can be added in the future.
- Effects or transformations may be applied to one or more layers. Currently, black and white, chroma and lumma key effects are required but others can be demanded in the future.
- Some UIs are required to interact with the application, to automatically change the configuration by means of the scheduler and to communicate SMS messages to Flash layers.
- A product must work 24 hours, seven days a week and cannot stop broadcasting. Therefore, any update or change must be performed without affecting the broadcasting.

From this information, we build a feature model which is depicted in Figure 2. Next, we follow the 4 steps to produce a component model from the feature model:

Step 1: Defining the core architecture

To define the core architecture, we need to determine the core set of features. In our case, *TV Platform*,*Layers*,*Effects*,*User Interface* and *Basic UI* features will compose the core architecture.

A component is created for each feature and they are connected by dependencies that come from `R1`, `R2`, `R3` and `R5` relationships.

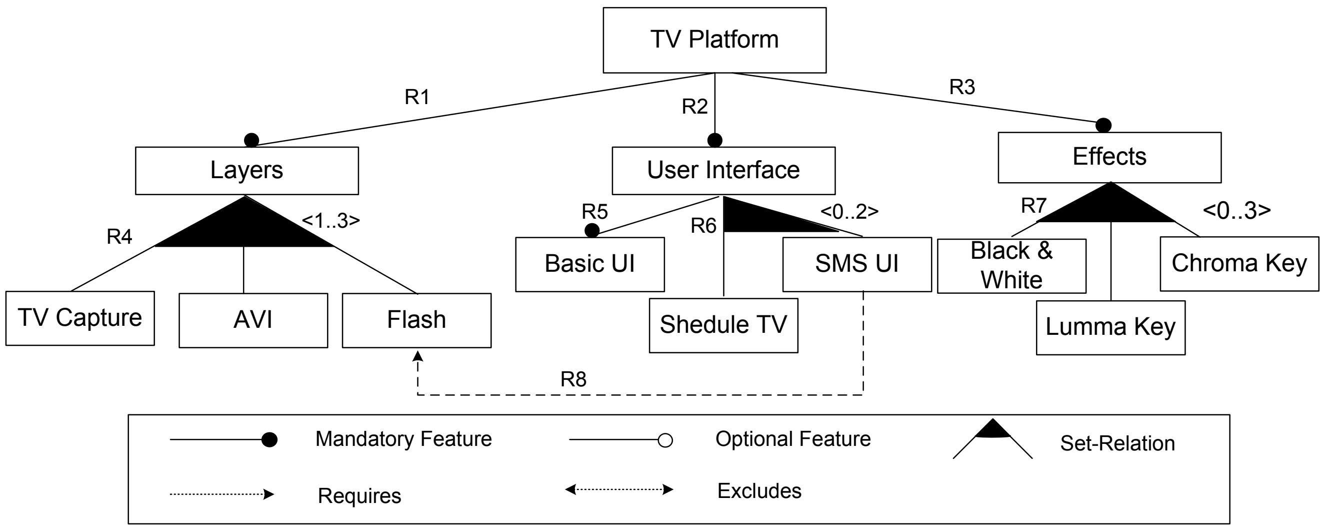


Figure 2. Feature Model describing the TV platform SPL case study

Step 2: Defining the dynamic architecture

For each remaining features, a feature component is created that provides the `Feature` interface. The relationship components must be added now. `R4`, `R6`, `R7` and `R8` relationship components are created and connect the respective feature components. As `R4`, `R6` and `R7` relationships in the feature model link a core feature with a non–core one, they are considered as part of the core architecture.

Step 3: Adding the configurator

The configurator component is added and coupled with the non–core features. As it can be seen, the configurator is not coupled with the relationships, as it is responsibility of the feature components to communicate them any change in the configuration.

The component model is generated at this point and depicted in Figure 3. Although it is not depicted in the component model, it is important to remark that the `Basic UI` component is in charge of de/activating any non–core feature and controlling any update of the feature components.

Step 4: Defining the initial product

Depending on the customer, the initial product have to be defined. For example, we can just activate the features needed to broadcast a TV signal with no effect, that will imply to initially activate only the `TV Capture` component.

Case Study Conclusions

In the resultant component model, if a customer demands a new kind of layer, only a component that implements the `Layer` interface must be developed. However, this change will affect the feature model by adding a new feature in the `R4` relationship, so the configurator component must be conscious of that change. This process will be analogous for effects and user interfaces.

5 Feature Models Analysis Operations

In any resultant model following the proposed mapping, the configurator component plays a determinant role in the dynamic behaviour of a product. This component must know the SPL feature model and extract relevant information to make decisions. From the responsibilities assigned to the to the configurator component and the operations needed for the mapping process, we determine the analysis operations on feature models that we need, and how current proposals give solutions to them:

- Determining the core–assets: in the first step to produce the component model, it is necessary to determine which are the features that compose the core architecture. Commonly, the core architecture is composed by those features that are shared by every product. However in some cases there are features which commonality is so high that it is interesting to consider them to be core features. In [13] a solution is given for this operation supported by the commonality factor which calculus and implementation based on constraint satisfaction problems(CSP) is described in [4].

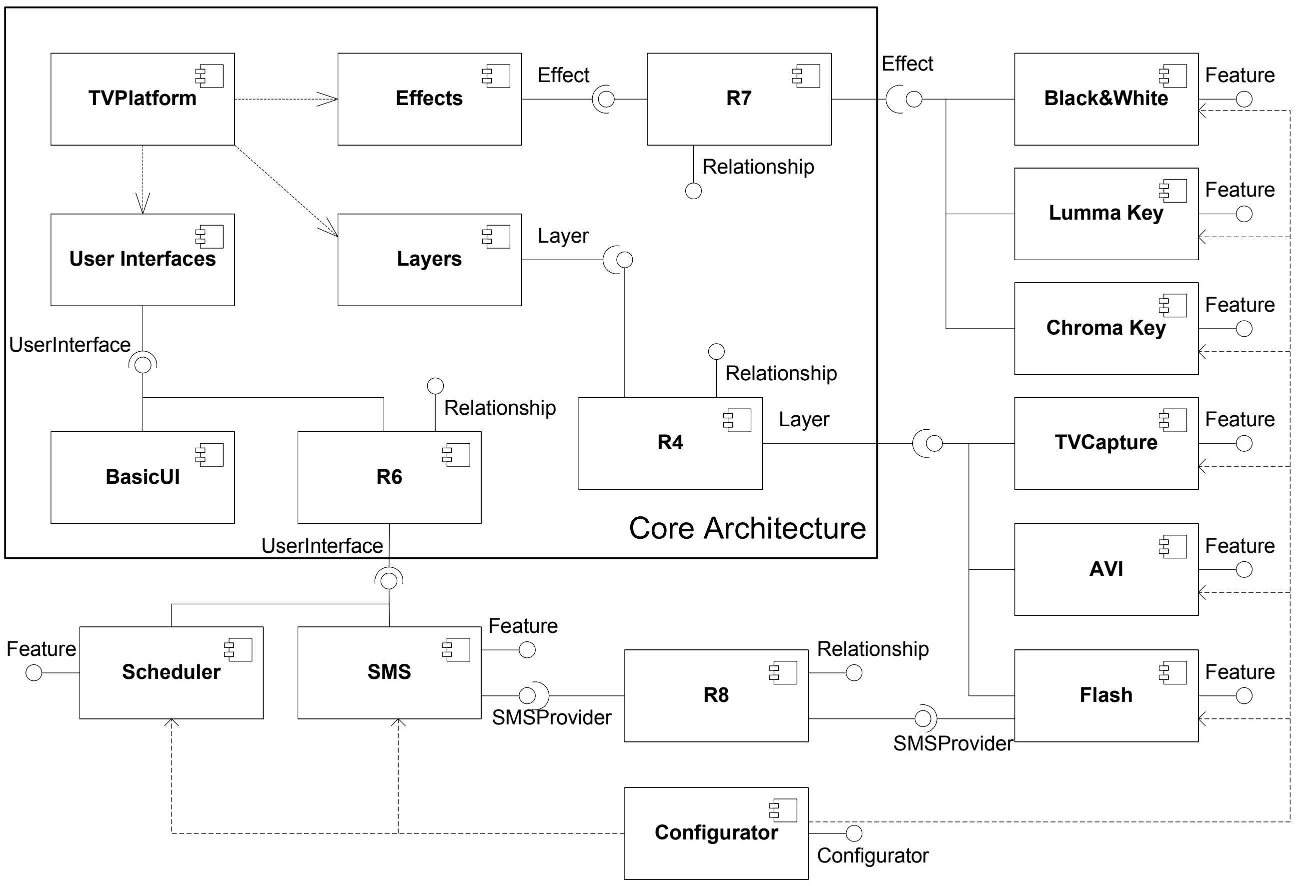


Figure 3. Component model for the TV platform SPL case study

- Determining if a product is valid: either for the initial configuration of a product or for any change suggested to the configurator, it must be checked whether it is possible to configure the product with the demanded features. This analysis operation is described by Benavides *et al.* [4] and an implementation that uses CSP solvers is proposed.
- Propagating decisions: when a feature is de/activated it usually has consequences for other features and relationships. To determine which are the features affected, Logic Truth Maintenance Systems(LTMS) and SAT solvers can be used to propagate feature de/activation[1].
- Explanations: in some cases, some features are demanded to be de/activated but the resultant product is not valid. In products such as real–time or critical systems it is important to de/activate a feature independently from the current configuration. The configurator must de/activate the features and reach a valid product. Explanations are used to determine which are the additional features that must be de/activated to become the product valid. This operation is firstly described in [2] and implemented to detect and explain modeling errors in feature models in [15]. The proposed implementation that uses CSP solvers, could be adapted to support the fore mentioned case of explanation.

These operations implementation relies on both CSP and SAT solvers to implement the solutions. The configurator could make use of the multiparadigm structure of FAMA tool [5] to support all these operations. However, the quickest response is needed for every operation, as they must be performed at run–time. As commented in Section 6, it is needed a benchmark of current implementations to determine the best techniques and algorithms that could be integrated into *Configurator* component.

6 Conclusions and Future Work

A process to automatically build a component model from a feature model is proposed based on the assumption that a feature can be modeled as a component. This approach can be useful in a service-oriented SPL where each component corresponds to a web service or to ubiquitous systems where each component corresponds a service provided by several sensors and actuators.

We envision that relationship components can be described as white–boxes whose implementation depends on the kind of relationship in the feature model. In [6] we implement each set-relationship component applying a dynamic abstract factory design pattern that we think that can be used in other contexts. Our future work will head for proposing transformation patterns with a higher complexity.

We remark the necessity of an automated support for some analysis operations on feature models. Currently, there exist solutions for some of operations, but others still need a thorough study to provide a real–time response. Our future work will focus on extending our current feature model analysis(FAMA) framework [5] with the operations that we have demanded in this paper. Furthermore, a quick response for the analysis operations is needed. We need benchmarks to determine the techniques and algorithms with the best performance so we could integrate them in the configurator component to provide real–time responses.

References

[1] D. Batory, "Feature models, grammars, and propositional formulas." in *Software Product Lines Conference, LNCS 3714*, 2005, pp. 7–20.

[2] D. Batory, D. Benavides, and A. Ruiz-Cortés, "Automated analysis of feature models: Challenges ahead," *Communications of the ACM*, vol. 49, no. 12, pp. 45–47, 2006.

[3] D. Batory, J. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement." *IEEE Trans. Software Eng.*, vol. 30, no. 6, pp. 355–371, 2004.

[4] D. Benavides, A. Ruiz-Cortés, and P. Trinidad, "Automated reasoning on feature models," *LNCS, Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005*, vol. 3520, pp. 491–503, 2005.

[5] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Corts, "FAMA: Tooling a framework for the automated analysis of feature models," in *Proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*, 2007, pp. 129–134.

[6] J. Bermejo, P. Trinidad, D. Benavides, and A. Ruiz-Corts, "Telvent: System families variability management using design patterns," in *Software Product Lines in Action*, F. van der Linden, K. Schmidt, and E. Rommes, Eds., vol. por determinar. Springer–Verlag, 2007.

[7] M. Bernardo, P. Ciancarini, and L. Donatiello, "Architecting families of software systems with process algebras," *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 4, pp. 386–426, 2002.

[8] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Techniques, and Applications*. Addison–Wesley, may 2000, iSBN 0–201–30977–7.

[9] S. Jarzabek, W. C. Ong, and H. Zhang, "Handling variant requirements in domain modeling," *The Journal of Systems and Software*, vol. 68, no. 3, pp. 171–182, 2003.

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

[11] K. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, "FORM: A feature–oriented reuse method with domain–specific reference architectures," *Annals of Software Engineering*, vol. 5, pp. 143–168, 1998.

[12] D. Liu and H. Mei, "Mapping requirements to software architecture by feature-orientation," pp. 69–76, 2003.

[13] J. Pea, M. Hinchey, A. Ruiz-Corts, and P. Trinidad., "Building the core architecture of a multiagent system product line: With an example from a future nasa mission." in *7th International Workshop on Agent Oriented Software Engineering*. LNCS, 2006.

[14] P. Sochos, M. Riebisch, and I. Philippow, "The feature-architecture mapping (farm) method for feature-oriented development of software product lines." in *ECBS*. IEEE Computer Society, 2006, pp. 308–318.

[15] P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro, "Automated error analysis for the agilization of feature modeling," *Journal of Systems and Software, in revision*, 2006.