A Model-driven Approach for Software Product Lines Requirements Engineering.

Conference Paper · January 2008

Source: DBLP

7 authors, including:

João Santos
New University of Lisbon
12 PUBLICATIONS **94** CITATIONS
SEE PROFILE

Ana Moreira
New University of Lisbon
220 PUBLICATIONS **2,950** CITATIONS
SEE PROFILE

João Araújo
New University of Lisbon
209 PUBLICATIONS **2,542** CITATIONS
SEE PROFILE

Vasco Amaral
New University of Lisbon
129 PUBLICATIONS **1,067** CITATIONS
SEE PROFILE

A Model-Driven Approach for Software Product Lines Requirements Engineering

Mauricio Alférez, Uirá Kulesza, André Sousa, João Santos,
Ana Moreira, João Araújo, Vasco Amaral
Dept. Informática, FCT, Universidade Nova de Lisboa, Portugal
{mauricio.alferez, uira, als, jps, amm, ja, vasco.amaral}@di.fct.unl.pt

Abstract

UML and feature models complement each other well and can be the base techniques for a systematic method to identify and model software product line (SPL) requirements. In this paper, we present a model-driven approach to trace both features and UML requirements analysis model elements, and to automatically derive valuable models for domain and application engineering. The resulting contribution is a synergetic approach for SPL requirements. We illustrate it by using a home automation system product line.

1. Introduction

Software product line (SPL) approaches [1-3] aim at improving the productivity and quality of software development by enabling the management of common and variable features of a system family. A system family is defined as a set of programs that shares common functionalities and maintains specific functionalities that vary according to specific family product members. A SPL can be seen as a system family that addresses a specific market segment [1].

Over the past few years, several SPL development approaches have been proposed [1-4]. Most of them motivate the identification of common and variable features of the SPL by means of domain analysis activities. A feature [4] can be seen as a system property or functionality that is relevant to some stakeholder and is used to capture commonalities or discriminate among products in a SPL. SPL features are typically represented in domain analysis using feature models [5]. Other requirements models (e.g., use case and activity models) can be used to better describe and detail the SPL requirements. The feature and requirements models are then used as a reference along all the process to guide the development of the SPL.

Some research works have addressed the use of feature models in combination with other models. Approaches like [6] and [7] propose to create relationships between features and UML models by means of intrusive graphical elements such as, presence conditions or notes to indicate variability. The main disadvantage of these approaches is the creation of convoluted and polluted models, which bring difficulties to understand, maintain and scale the models and trace links between features and UML elements.

Other approaches [3, 8, 9] give some directions on how to model and trace variability information. However, and similarly to what happens with the previous approaches, they do not provide specific activities and tool support for modeling, tracing and generating requirements models for specific products based on the tracing information.

This paper presents a model-driven approach for variability management in product lines that addresses traceability between features and UML requirements models (like use cases and activity models). The main contribution is to show how model-driven techniques can be used to automatically derive, from the information provided by the trace links, requirements models for specific products of a SPL, and views that explicitly illustrate the relationships between features and UML requirements model elements. These views are useful in both domain and application engineering stages. The general idea of our approach is to apply bindings between metamodels, create a simple tracing metamodel strategy, generate specific product requirements models automatically, and use composition rules to specify compositions between use cases by means of their respective activity diagrams.

This paper starts with an overview of our metamodelling strategy and approach main activities in

Section 2. These activities are illustrated using a home automation system, in Section 3. Section 4 explores and presents lessons learned from the application of our approach. Finally, Section 5 concludes the paper and points out directions to some future work.

2. A Model-Driven Approach for SPL Requirements Engineering

Traceability between feature and requirements models is supported in our approach by a metamodelling strategy. Figure 1(a) introduces the adopted metamodelling strategy and Figure 1(b) makes that strategy concrete through feature, use case and activity metamodels.

A variability model is used to represent the common and variable SPL features. One or more requirements models detail the SPL requirements. A traceability metamodel is used to link abstractions from the variability and the requirements models. This enables the navigation across abstractions of the different types of models using model-driven techniques and tools. The traceability model also supports backward and forward traceability between a feature model (or any of its configurations) and requirements models. Each configuration defines the features of a specific product from the SPL.

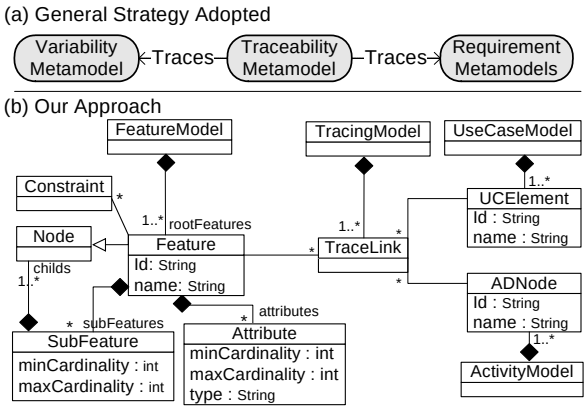

Figure 1. Traceability support strategy

Our approach adopts a feature metamodel based on [7] as the variability model. UML use case and activity models specify the SPL requirements. Due to the large dimension of the their metamodels, we only show the use case model element "UCElement" and activity diagram node "ADNode" from which all the traceable elements of each model can be inherited. Activity diagrams model the behavior of use cases. Use cases and activity models are related to each other by means of the feature to which they are connected.

Our metamodel (Figure 1(b)) supports the set of models that we create in domain and application engineering. The metamodel enables the creation of models in *conformance to* their respective metamodels [10], and helps to understand the relationships between the models elements. Besides the metamodelling strategy, our approach also defines a set of systematic activities in the domain and application engineering stages. The SPL requirements models are created and manipulated during these stages using model-driven techniques and tools.

At the domain analysis level, we perform the activities described next. Although they are organized sequentially, they are typically executed iteratively and incrementally.

1. Identify requirements. The SPL requirements can be elicited using traditional requirements engineering techniques such as inspection of existing documents that describe the problem domain, existing catalogues [11], stakeholders interview transcripts or by using mining techniques [12]. Other approaches such as [9] and [3] already address this activity in detail.

2. Group requirements into features. During this activity, we organize the SPL requirements into clusters according to the specific SPL features they are related to. There are semi-automatic clustering techniques such as [13] that could help to support this activity. However, the specific steps followed in the clustering sub-process are out of the scope of this paper and are not included due to lack of space.

3. Refactor requirements and features. During the previous activity, requirements could result to be linked to more than one feature. We propose to refactor those requirements to be ideally related to only one feature, whenever possible. It contributes to achieve a better modularization of the SPL requirements through the separation of the variable parts of each requirement [14] as well as facilitate establishing tracing links between requirements and features.

4. Model SPL features and use cases. This activity structures and represents the SPL requirements using use case and feature models. Use case models specify the functional requirements and feature models specify the SPL features and variability-commonality information.

5. Relate features to use cases. The relationships between features and use cases are specified visually in a table of trace links. The table allows defining and maintaining the trace relationships between features and UML elements.

6. Generate SPL use cases annotated with features. A model-driven tool developed for our approach uses the relationships between use cases and features to generate specific use case models annotated with features [15]. In the annotated model, each use case is shown with the respective(s) feature(s) related to it. Therefore, it is also possible to obtain the set of use cases related to a specific feature. This allows the domain analysis engineers and SPL architects to reason about how each use case is related to the SPL features and to analyze the impact of change of specific features in SPL requirements.

7. Model use cases as activity diagrams. The detailed behavior of each use case is modeled using activity diagrams, similarly to what happens in several UML-based methods, such as RUP [16]. Use cases specified as activity diagrams, in contrast with textual-based specifications, allows us to enable the use of model-driven generation tools by providing models that conform to a metamodel (i.e., UML activity diagram metamodel) and to help to avoid ambiguity in the specifications [3]. The detailed specification of use cases as activity models also enables us to customize the behavior of use cases according to the features selected to a specific SPL configuration.

8. Specify composition rules between use cases. Each composition rule defines how a variable use case (i.e., linked to a variable feature) can interfere or modify the normal execution of a mandatory use case (i.e., linked to a common feature). Composition rules are defined in terms of the elements of the activity diagrams (e.g., activities, initial state or final state).

The models produced during domain engineering are used in application engineering to generate use case and activity models for specific SPL configurations. We define three activities in application engineering:

1. Define a SPL configuration. The application engineer specifies a SPL configuration, where s/he chooses which optional and alternative features are going to be part of the final application.

2. Generate a use case model from a SPL configuration. Our tool [15] generates the use case model related to the SPL configuration defined in the previous activity. The input for the generation is the SPL use case model, the SPL configuration and the table that maintains the trace links between features and use cases.

3. Generate activity diagrams from a SPL configuration. Our tool is also used to generate activity diagrams related to a specific SPL configuration. In this process, the original activity diagrams can be composed using the composition rules defined in the domain engineering stage. The choice of which composition rules will be used is based on the features included in the SPL configuration. The activity diagram of each extension use case, for example, can be composed with mandatory use cases if the variable feature related to it was selected by the application engineer (step 1 of application engineering).

3. Applying the Approach to a Case Study

To illustrate the activities described in the previous section, we have chosen a home automation system, called Smart Home (see also [3]). This system is one of the SPL case studies proposed by the industrial partners of the European project AMPLE [17]; due to its complexity, we will focus only on a subset of the Security module.

The requirements and feature identification, and refactoring activities, are described in [18]. They provided the features and requirements of our case study. By inspecting those requirements and features, we modeled the SPL feature and use case models. Figure 2 shows the most relevant artifacts produced by the activities of our approach. It shows how each artifact produced in the domain engineering perspective is used to create or derive other artifacts for a specific product in the application engineering perspective. Next we describe the domain engineering activities from our approach.

Model SPL features and use cases. Figure 2(a) shows the feature model of our Security module. It has three main features: *Room Surveillance*, *Admittance Control* and *Intrusion Detection*. *Room Surveillance* is an optional feature that includes *Indoor Camera Surveillance* and, optionally, *Indoor Motion Detection*. The inhabitant can be admitted to enter the house after passing either a *Biometrical Analysis*, *Smart Card*, or entering a *PIN*. In case of selecting intrusion detection, the *Glass Break Detection* must be included and optionally, motion detection sensors and cameras for outdoor security. The notation used in Figure 2(a) is described in Figure 2(j).

We can obtain the SPL use cases from the requirements and features previously identified. Use case modeling is used to better structure the SPL requirements and add more semantics to the features [6]. Figure 2(b) shows the use case model of the case study. The initial SPL features and use cases can be refined and incremented to consider new variabilities or products that need to be included in the family. Both use case and feature models must be updated when

new features are considered or existing ones need to be modified or removed.

Relate features to use cases and generate SPL use case models annotated with features. So that traceability can be maintained between use cases and features, we define an activity to specify the trace relationships between those artifacts. By inspecting the requirements and features of the case study, we related, for example, the *Open Front Door* use case with *Admittance Control*, refined into *Biometrical Analysis*, *Smart Card*, and *PIN* features because to open the front door, the system requires *Admittance Control.* Figure 2(c) shows one of the views that can be generated using the trace links relationships between use cases and features. The open branch in the tree-like structure shows that the *Smart Card* feature is related with the use cases *Identify User by Smart Card*, *Open Front Door* and *Configure Security Management*.

The traceability views of the relationships between features and other artifacts allow the domain analysis engineers and SPL architects to reason about the domain analysis artifacts interdependencies. Currently, there are two kinds of traceability views that our approach can generate in this activity: (i) A use case model annotated with the respective related features; and (ii) a tree structure that shows the list of use cases with the related features and, optionally, the list of features with the related use cases (as in Figure 2(c)).

Create activity diagrams. The behavior of each use case can be specified in our approach using activity diagrams. These diagrams were created by inspecting the requirements. Figure 2(e) and Figure 2(f) shows, for example, the activity diagrams of the *Identify User by Smart Card* and *Identify User* use cases.

Specify composition rules between use cases. Use cases composition is addressed in our approach by means of a set of composition rules. Each composition rule defines how a use case can interfere, modify, or replace the execution of another use case. The composition rules are defined in terms of activity diagrams elements (i.e., activities, initial and final nodes). Composition rules are used during the application engineering phase to derive the specific behavior of use cases for a SPL configuration or product. Figure 2(d) presents the composition rule between the use cases *Identify User* and *Identify User by Smart Card*. It shows how the *Identify User by SmartCard* use case can modify the *Identify User* use case to include additional steps related to the *Smart Card* variable feature. The application of the composition rule is shown in the following subsection where specific activity diagrams can be generated for each product of the SPL.

Next, we describe the execution of the application engineering activities of our approach in the context of the Smart Home case study.

Define a SPL configuration and generate the related use cases and activity diagrams. The first activity in application engineering is to specify a SPL configuration to decide which features will be part of the final application. Figure 2(g) shows a configuration of the case study feature model shown in Figure 2(a) (see the notation used in Figure 2(j)).

Based on the feature model configuration, the relationships between use cases and features, and the SPL use case model (Figure 2(b)), a use case model can be automatically derived using the tool from our approach [15]. Figure 2(h) shows the use case model of the product specified in Figure 2(g).

The final activity of application engineering in our approach involves the automatic customization of the activity diagrams related to each of the SPL use cases using the composition rules specified in domain engineering. Only the activity diagrams of the use cases that are part of the SPL configuration are customized. Figure 2(i) shows the composition between the activity diagrams that describe the *Identify User* and *Identify User by Smart Card* use cases depicted in Figure 2(f) and Figure 2(e), using a `Replace with` composition rule depicted in Figure 2(d). It is not the aim of this paper to present a full-fledged composition language; we just show how it would look like. A complete composition language is one of our aims for future work. For additional details about the current version of our composition language, please refer to [17].

4. Benefits and Lessons Learned

In the context of the European project AMPLE, experiments with our approach have shown that the information of the relationships among the SPL requirements models can be used to support: (i) forward and backward traceability between features and requirements models like use case and activity models; and (ii) reasoning about the impact of feature interactions in the SPL requirements (expressed by the use cases and activity diagrams). Forward and backward traceability enables the creation of tracing queries over all the requirements artifacts and the derivation of specific requirements models for a determined product in the SPL using an model-driven derivation tool, as the one that we have developed [15]. In addition, it enables to the developers to visualize the features changes effects in the SPL requirements through the automatic modification of the

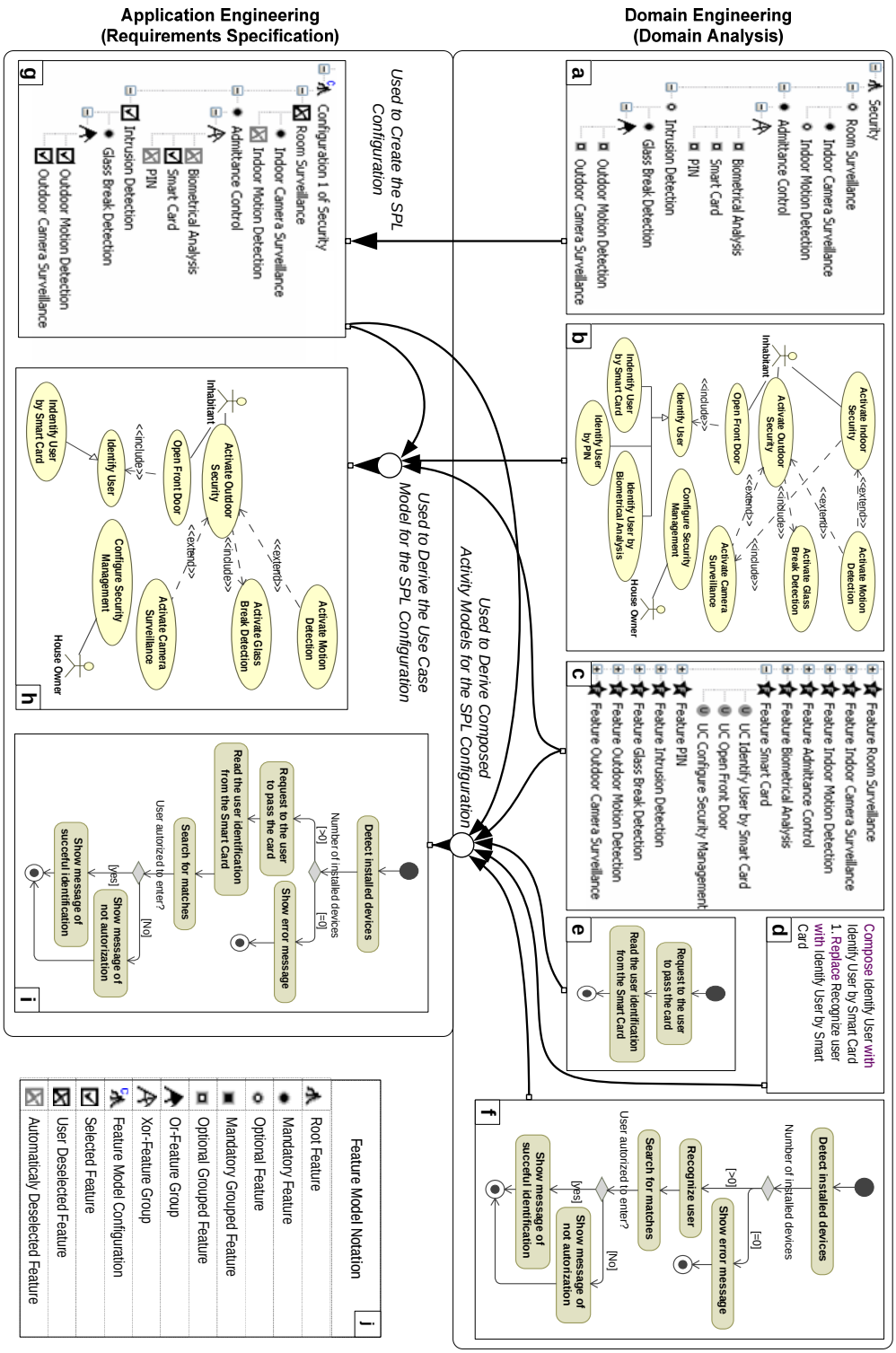

Figure 2. Some of the artifacts produced in the Smart Home Security module case study. (a) SPL feature model; (b) SPL use case model; (c) Use cases related to features; (d) Composition rule between "Identify User" and "Identify User by Smart Card"; (e) Activity diagram of the "Identify User by Smart Card" use case; (f) Activity diagram of the "Identify User" use case; (g) Configuration of the SPL feature model; (h) Use case model for a specific product; (i) Composing "Identify User" with "Identify User by Smart Card"; (j) Feature model notation.

models. On the other hand, the information about feature interactions offered by our approach is useful during the design of SPL architectures to allow an adequate modularization and implementation of their respective features. However, in this paper we have only concentrated on describing the traceability functionalities.

Our metamodelling strategy (Section 2) brings the following benefits to the definition of our approach: (i) *simplicity* – the integration between the metamodels of the feature and requirements models is very easy to understand and evolve; and (ii) *flexibility* – the strategy can be applied to any requirements notation that has a well-defined metamodel.

Our approach also enables composition of crosscutting use cases by representing their steps in activity diagrams. Composition rules are used to specify how the behavior of a use case can affect the behavior of another one. We believe this is an effective way to represent how the SPL variabilities occur along the use cases behavior. The integrated use of these activity diagrams, composition rules and a SPL configuration allows generating the specific behavior of a SPL product. The resulting activity diagram representing the use cases of a product can then be used with different purposes, such as, for example, to document the final requirements of the product or to generate specific test cases for the product.

5. Conclusions and Future Work

This paper presented a model-driven approach to model, specify and trace SPL features and requirements supported by an automated tool. We adopted a simple but useful metamodel integration strategy to allow tracing between features and other requirements models. The approach includes domain and application engineering activities, both illustrated using the Smart Home SPL case study.

Our work is currently being extended to address additional perspectives, such as: (i) to provide more explicit guidance for non-functional requirements and feature interactions modeling and to create special trace views for these concerns; (ii) to deal with uncertainty or volatile requirements in SPLs; (iii) to continue exploiting the activity diagrams to model scenarios [19]; and (iv) define a more complete approach in the context of the AMPLE project to provide tracing support from features and requirements models to artifacts of later software development stages, such as, architecture models and source code. Finally, a full-fledged composition language will be defined.

Acknowledgements. The authors are partially supported by EU Grant IST-33710: Aspect-Oriented, Model-Driven Product Line Engineering (AMPLE).

References

[1] P. Clements and L. M. Northrop, *Software Product Lines: Practices and Patterns*. Boston, USA: Addison-Wesley, 2002.
[2] D. M. Weiss and C. T. R. Lai, *Software Product-line Engineering: a Family-based Software Development Process*. Boston, USA: Addison-Wesley, 1999.
[3] K. Pohl, et al, *Software Product Line Engineering: Foundations, Principles and Techniques*. Berlin, Germany: Springer, 2005.
[4] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*: ACM Press/Addison-Wesley, 2000.
[5] K. Kang, et al, "Feature-Oriented Domain Analysis (FODA) Feasibility Study", SEI, CMU/SEI-90-TR-021, 1990.
[6] K. Czarnecki and M. Antkiewicz, "Mapping Features to Models: A Template Approach Based on Superimposed Variants", presented at GPCE, Tallinn, Estonia, 2005.
[7] A. Bragança and R. J. Machado, "Automating Mappings between Use Case Diagrams and Feature Models for Software Product Lines", presented at SPLC, Kyoto, Japan, 2007.
[8] H. Gomaa, *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*: Addison-Wesley, 2004.
[9] M. L. Griss, et al, "Integrating Feature Modeling with the RSEB", presented at ICSR, 1998.
[10] J. Bézivin, "On the Unification Power of Models", *Software and Systems Modeling*, vol. 4(2), pp. 171-188, 2005.
[11] L. Chung, et al, *Non-Functional Requirements in Software Engineering*, 1 ed: Kluwer Academic Publishers, 1999.
[12] A. Sampaio, et al "EA-Miner: A Tool for Automating Aspect-Oriented Requirements Identification", in *Proceedings of ASE*, Long Beach, CA, USA, ACM Press, 2005, pp. 352-355.
[13] K. Chen, et al, "An Approach to Constructing Feature Models Based on Requirements Clustering", in *Proceedings of RE*, Paris, France, IEEE Computer Society, 2005, pp. 31-40.
[14] A. Moreira, J. Araújo, and J. Whittle, "Modeling Volatile Concerns as Aspects", presented at CAiSE, Luxemburg, Luxemburg, 2006.
[15] "AMPLE Project Research Group at FCT/UNL", http://ample.di.fct.unl.pt/.
[16] P. Kruchten, *The Rational Unified Process: An Introduction*: Addison-Wesley, 2003.
[17] AMPLE, "Ample Project", http://www.ample-project.net/.
[18] M. Alférez, et al, "Traceability between Features and UML-Based Requirements Models: A Model-Driven Approach for Product Lines Engineering", http://ample.di.fct.unl.pt/tool/Alferez-etal-TR-1-2008.pdf
[19] N. Maiden and I. Alexander, *Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle*: John Wiley & Sons, 2004.