Feature-Oriented Model-Driven Software Product Lines: The TENTE approach*

Lidia Fuentes, Carlos Nebrera, and Pablo Sánchez

Dpto. Lenguajes y Ciencias de la Computación
Universidad de Málaga, Málaga (Spain)
{lff,cnebrera,pablo}@lcc.uma.es

Abstract. In recent years, modern techniques for advanced separation of concerns and Model-Driven Development (MDD) have provided new means for improving the current methods of Software Product Line (SPL) Engineering. Mechanisms such as *family polymorphism* and *mixin composition* can be used to improve the separation and composition of features of an SPL. Using MDD repetitive, laborious and time consuming tasks of SPL processes can be automated. Nevertheless, there is a general lack of SPL processes that integrate advanced mechanisms for separation of concerns with MDD techniques. This paper presents an innovative process, called TENTE, which combines both technologies. The result is a model-driven process that maintains the separation of features both at the architectural design and implementation stages, improving separation of variants; at the same time repetitive, laborious and time consuming tasks are automated.

1 Introduction

A Software Product Line(SPL) [1] aims to create the infrastructure for the rapid production of software systems for a specific market segment. These software systems share a subset of common features, but variations are also present. Software Product Line Engineering involves two new issues as compared to engineering of single software-based systems: *variability design* and *product derivation*.

Variability design is concerned with incorporating variation mechanisms into the software products, which enable the construction of an infrastructure representing a complete range or family of products. Such an infrastructure will include both the commonalities and variations of the family of produts. *Product Derivation* is the process of constructing specific software products, after a specific configuration (i.e. a valid set of alternatives and variants) has been selected.

Modern software decomposition techniques [2], such as *family polymorphism* and *mixin composition*, provides new mechanisms for the separation and composition of concerns. These mechanisms can be applied to *variability design*,

* This work has been supported by Spanish MCYT Project TIN2008-01942 and the EC STREP Project AMPLE IST-033710.

improving the separation and composition of variable features in an SPL [3]. Model-Driven Development offers mechanisms for automating repetitive and time consuming tasks of the SPL development lifecycle, such as *product derivation* processes [4].

Currently there is no process for architectural design and implementation that: (1) uses advanced techniques for separating variable features at the architectural design and implementation stages; (2) uses model-driven techniques for automating repetitive, laborious and time-consuming tasks; and (3) generates automatically models of specific products for each development stage. This latter issue is helpful for reengineering specific products according to user requirements not originally covered by an SPL.

This paper presents as main contribution a feature-oriented model-driven process, named TENTE[1], for SPL architectural design and implementation. This process integrates relevant advances, from an SPL point of view, for separation of concerns and MDD technologies. Advanced mechanisms for separation of concerns enable the encapsulation of variants in separate units. This separation simplifies variant management and composition, thereby facilitating product derivation. Separation of variants is kept both at the architectural design and implementation levels. Moreover, MDD techniques help to automate part of this process, such as the generation of the implementation skeletons or the product derivation process, avoiding the need for repetitive and tedious tasks to be performed manually. For each specific product derived from the SPL, a software architecture and an implementation, specific for that product, are obtained. Moreover, this process does not require any knowledge of model-driven techniques

After this introduction, this paper is structured as follows: Section 2 describes the different steps that comprise our approach. Section 3 discusses the benefits it provides, and concludes with comments on related and future work.

2 The TENTE approach

This section provides a general overview of TENTE. The process is comprised of five steps, as depicted in Figure 1. It covers the architectural design and implementation software development stages, both at the domain and application engineering levels. Software architectural models are expressed in UML 2.0. The implementation language selected is CaesarJ [2], a language with special features, such as *virtual classes* and *mixin composition*, for Feature-Oriented Programming (FOP).

The first three steps correspond to the Domain Engineering level. They serve to create the infrastructure from which specific products will be derived. The last two steps correspond to the Application Engineering level and they serve to

[1] TENTE is the Spanish name for Lego. We have selected this name because we view an SPL as a Lego game: it is about constructing specific products from prebuilt blocks.

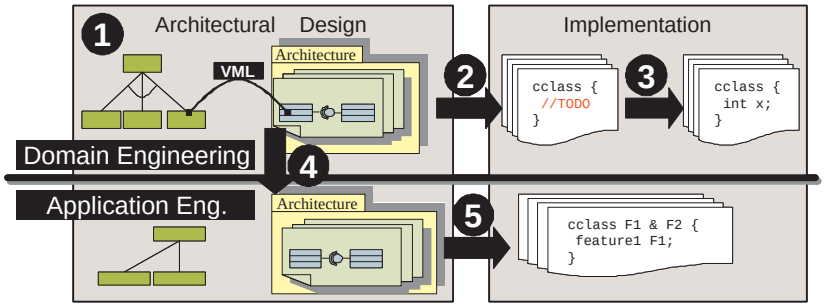

Fig. 1. The TENTE approach

create specific products inside an SPL. The whole process is as described in the following subsections.

2.1 Domain Engineering

Step 1: Architectural Design

First of all, an architectural model for the SPL is constructed (Figure 1, label 1). This architectural model is comprised of three elements: (1) a cardinality-based feature model; (2) a UML 2.0 model; and (3) a VML (*Variability Modelling Language*) specification.

The cardinality-based feature model [5] specifies which parts of the architecture are variable and why they are variable. This represents *problem space* or *variability specification*.

The UML 2.0 model, which we have named *reference architecture*, contains the architectural design of both the commonalities and the variabilities of a complete family of products. Coarse-grained variants are separated in different UML packages, which are then combined by means of the UML *merge* operator, similarly to Laguna et al [6]. Each package represents an architectural increment, which adds new components, interfaces and so forth to an existing architecture, extending it with new functionalities. Fine-grained variants are supported using traditional techniques, such as the same interface being implemented by different components. For modelling the software architecture, component, class, composite structure, deployment and sequence diagrams are used. At domain engineering level, component types for constructing specific products are specified. At application engineering level, instances of these component types are assembled for configuring specific products.

Figure 2 shows an example of feature and software architectural models for a Smart Home case study, provided by Siemens in the context of the AMPLE project[2].

[2] http://www.ample-project.net

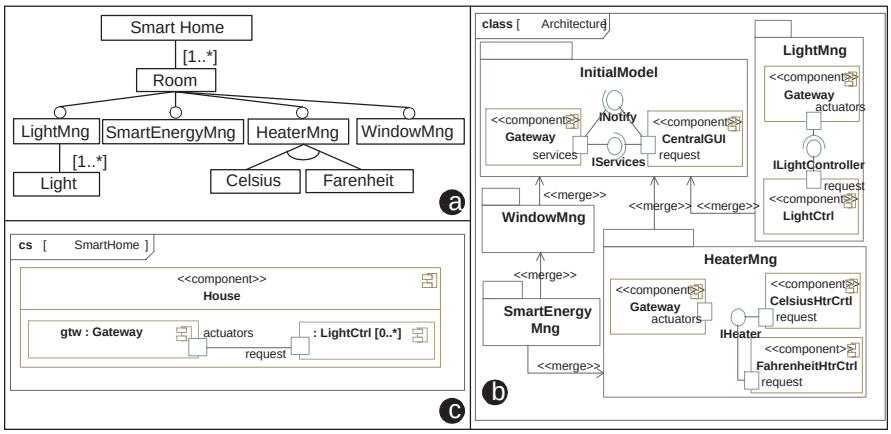

Fig. 2. Architectural design at domain engineering level

```
00 VariationPoint HeaterMng {
01   Kind Alternative;
02     Variant Celsius{
03        SELECT: connect(Gateway,CelsiusHtrCtrl) through interface(IHeater);
04        UNSELECT: remove(CelsiusHtrCtrl);
05     }
06     Variant Fahrenheit{
07        SELECT: connect(Gateway,FahrenheitHtrCtrl) through interface(IHeater);
08        UNSELECT: remove(FahrenheitHtrCtrl);
09     }
10 }
11 VariationPoint LightMng {
12    Kind Optional;
13    SELECT: merge(LightMng) into (InitialModel);
14    UNSELECT: remove(LightMng);
15 }
```

Fig. 3. VML specification for the Smart Home case study

The link between the feature model and the reference architecture is established using VML (*Variability Modelling Language*) [7] (see Figure 3), an innovative language for connecting variability specification (i.e problem space) with variability realisation (i.e. solution space). A VML specification also contains all the information required for automatically deriving the architectural model of a specific product from the reference architecture. For each variant, special primitives specify which actions must be carried out if a variant is selected or deselected. Figure 3 shows an excerpt of a VML specification for the example of Figure 2.

Step 2: Transformation of architectural models into implementation

Using a code generator, part of the implementation is automatically generated from the reference architecture (Figure 1, label 2). More specifically, the skeleton of components and the logic for connecting them are generated. The part corresponding to the behaviour of each method is left empty for the completion at the implementation level. Separation of variants achieved at modelling

level is preserved at the implementation level using CaesarJ *family classes* and *mixin composition* [2].

Step 3: Domain engineering implementation

Each component skeleton previously generated is completed with its corresponding business logic. As a result, a set of components implementing the family of products is obtained. We only need to appropriately instantiate and connect these components in order to obtain a specific product. This is addressed at the application engineering level. This step completes the domain engineering level.

2.2 Application Engineering level

At the application engineering level, a specific product is configured by selecting those features that must be included in that product and subsequently instantiating and connecting components according to that selection of features.

Step 4: Derivation of a specific architectural model

The first step in our process is the creation of a configuration of the feature model, i.e. a valid selection of variants to be included in a specific product. Using this configuration, the architectural model of the desired product is automatically derived from the reference architecture, by executing the VML specification. A VML specification is compiled into a set of model transformations that actually implement the product derivation process [7]. The main contribution of VML is that the software architect does not need to have any expertise in model transformation techniques, since the transformations are automatically generated by the VML compiler.

Step 5: Derivation of a specific implementation

The software architectural model obtained in the previous step is automatically transformed into a complete implementation in CaesarJ, using a code generator. This code generator basically creates the component instances which are required for assembling a specific product. These component instances are also appropriately initialised and connected by the code generator. As a result, the complete implementation of a specific product is obtained.

3 Conclusions and Future Work

This work has presented TENTE, a model-driven process for SPL architectural design and implementation. As compared to other approaches, TENTE provides:

1. Separation of coarse-grained variants both at the architectural and at the implementation levels, using UML packages combined by means of *merge*

operators and CaesarJ family classes, respectively. The separation of variants is therefore kept at the implementation level, unlike other model-driven approaches [8]. CaesarJ provides a stronger type system that enables feature instantiation and its polymorphic use. It also facilitates feature dependency management, compared to other feature-oriented approaches, such as Laguna et al [6] or Trujillo et al [4].
2. Several parts of the process are automated by means of model transformations. The application engineering level and product derivation processes are fully automated. At the domain engineering level, 30%-70% of the implementation code is automatically generated.
3. The generation of software architectural models at the application engineering level allows the software architecture of a specific product to be adapted according to new user requirements. Other approaches, such as Trujillo et al [4], only consider the creation of artefacts for specific products at the implementation level. Thus, the benefits of using models at different abstraction levels are lost at the application engineering level.

As future work, we have planned to add behavioural diagrams, such as UML 2.0 state machines for describing component protocols, to the the software architecture description. This will allow a larger amount of code to be directly derived from models, increasing the level of abstraction at which software systems are developed. We will also integrate the process presented in this paper with methodologies for SPL requirements engineering, in order to cover all the stages of the software lifecycle.

References

1. Pohl, K. et al: Software Product Line Engineering: Foundations, Principles and Techniques. Springer (2005)
2. Aracic, I. et al: An Overview of CaesarJ. In: Transactions on Aspect-Oriented Software Development I. (2006) 135–173
3. Mezini, M., Ostermann, K.: Variability Management with Feature-Oriented Programming and Aspects. In: 12th Int. Symp. on Foundations of Software Engineering (FSE). (2004) 127–136
4. Trujillo, S. et al: Feature Oriented Model Driven Development: A Case Study for Portlets. In: 29th Int. Conference on Software Engineering (ICSE). (2007) 44–53
5. Czarnecki, K. et al: Staged Configuration through Specialization and Multilevel Configuration of Feature Models. Software Process: Improvement and Practice **10** (2005) 143–169
6. Laguna. M. A. et al: Seamless Development of Software Product Lines. In: 6th Int. Conference on Generative Programming and Component Engineering (GPCE). (2007) 85–94
7. Sánchez, P. eh al.: Engineering Languages for Specifying Product-derivation Processes in Software Product Lines. In: 1st Int. Conference on Software Language Engineering (SLE). (2008)
8. Voelter, M., Groher, I.: Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In: 11th Int. Software Product Line Conference (SPLC). (2007) 233–242