Automating Product Derivation in Software Product Line Engineering

Goetz Botterweck
Lero, Univ. of Limerick
Limerick, Ireland
goetz.botterweck@lero.ie

Kwanwoo Lee
Hansung University
Seoul, South Korea
kwlee@hansung.ac.kr

Steffen Thiel
Lero, Univ. of Limerick
Limerick, Ireland
steffen.thiel@lero.ie

Abstract: This paper deals with deriving software products from a software product line (SPL) in an efficient and automated way. We present an approach that (1) represents the SPL with a set of integrated models, (2) specifies variability and configuration options for possible product variants and (3) automatically derives executable products with model transformations and aspect-oriented techniques. The approach is discussed with a sample SPL of scientific calculators.

1 Introduction

A software product line (SPL) is a set of software-intensive systems sharing a common, managed set of features that are developed from a common set of core assets in a prescribed way [CN02]. SPL engineering has rapidly emerged as a viable and important software development activity during the last few years. It allows companies such as Philips, Bosch, and Nokia to build functionally related products with a minimum of technical diversity and thus to achieve significant improvements in time-to-market, cost, productivity, and quality (e.g., [CN02, HFT04]). SPL engineering differs from developing single products in that variability is an inherent part of the development. Products are built by resolving this variability in order to implement customer-specific functionality. This is usually done in a dedicated product derivation process during application engineering.

Many companies attempt to achieve economies of scale in SPL engineering while keeping the number of product variants in their product lines high. Those large-scale product lines easily comprise hundreds of products and incorporate thousands of variation points and configuration parameters. This is especially the case in the mobile devices and automotive domain (e.g., [MH03, STB+04]). However, recent studies show that much of the work on product derivation is carried out manually (e.g., [DSB05]). This makes systematic product derivation extremely difficult, error-prone, and time consuming and compromises the benefits of product line adoption significantly.

In this paper we present an approach and research tool that support automated product derivation in software product lines. The approach integrates feature modelling and aspect-oriented programming (AOP) and is based on model-driven techniques. It provides the expressive means to describe a product line from two perspectives: (i) the variability and configuration options and (ii) implementation strategies for those options.

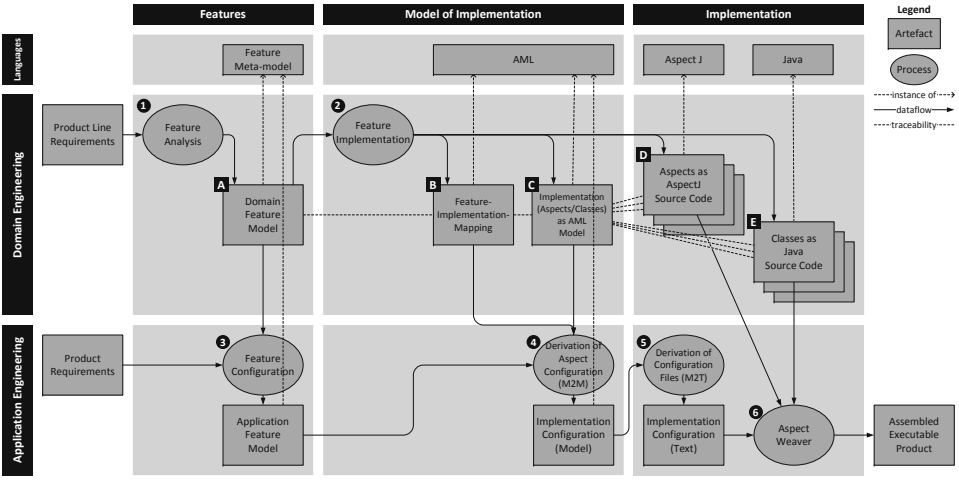

Abbildung 1: Overview of the presented approach.

The presented approach differs from existing research as it provides the explicit modelling of dependencies among features and the intentional separation of dependency implementations from feature implementations. This allows a direct mapping of feature dependencies to the corresponding dependency implementations which results in reduced coupling among feature implementation components, less side-effects, and, consequently, a more effective product derivation.

To facilitate the approach we have developed an interactive and visual derivation tool which supports the automated derivation of executable products based on a feature configuration. The remainder of the paper is structured as follows: Section 2 describes how we model a SPL, Section 3 explains how these models are used in our automated product derivation approach. In Section 4 our approach is compared with other related work. Section 5 concludes the paper.

2 Creating and Modelling a Software Product Line

In our approach the product line is described in terms of three models and two types of source code artefacts (see the markers **A** to **E** in Figure 1). These models and artefacts are created in a two-step process of Feature Analysis and Feature Implementation. We illustrate our approach based on a sample product line of calculator applications, which we created by refactoring the open source Java application *Java Scientific Calculator* [jsc08].

During *Feature Analysis* ❶ a *Domain Feature Model* **A** is created, which describes the capabilities of the SPL and potential functionality of products from a stakeholder's point of view. In addition, this model captures various dependencies among features: *Design-time*

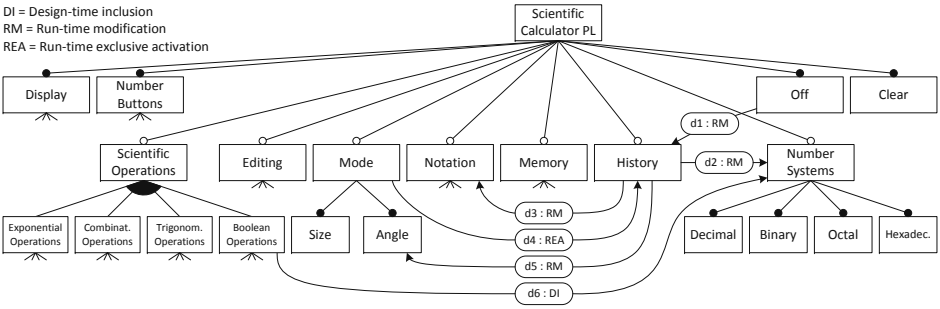

Abbildung 2: Feature Model (excerpt).

dependencies limit the available choices and guide the engineer during feature configuration ❸. *Run-time dependencies* describe the interaction among features in the executed product. Figure 2 shows the domain feature model of our Calculator SPL with mandatory features, e.g., `Display`, and optional features, e.g., `Editing`. It also shows some dependencies, some of which are design-time dependencies (d6=`BooleanOperations` requires `NumberSystems`) and some run-time dependencies (d5=`History` modifies the behaviour of `Angle`).

During *Feature Implementation* ❷ the capabilities of the software product line are implemented with *AspectJ* D and *Java* E. The *Implementation Model* C, described in our AML (Aspect-oriented implementation Modelling Language), provides an abstract view on this implementation. Just like the corresponding textual code, the AML model contains concepts like *Aspect* or *Class* but describes them as model elements. This allows to describe mappings B between features and the corresponding implementation components, which is necessary to select implementation units for a feature configuration.

Figure 3 shows some of these mappings for our Calculator example. For instance, the feature `History` is mapped to the package `history` and the dependency `d5` is mapped to the aspectual component `AngleHistoryDep.aj`.

3 Product Derivation

In the preceding section we discussed how the software product line is created and how the elicited knowledge is captured. This section describes how this knowledge is exploited during product derivation to create the assembled executable products.

Given the product-specific requirements, we start creating the product by first performing interactive *Feature Configuration* ❸. Since this activity involves the interpretation of requirements, it cannot be automated. However, it can be supported by interactive tools which provide visual guidance [BNP+07], e.g., by providing feedback based on constraints which have been captured in the Domain Feature Model A.

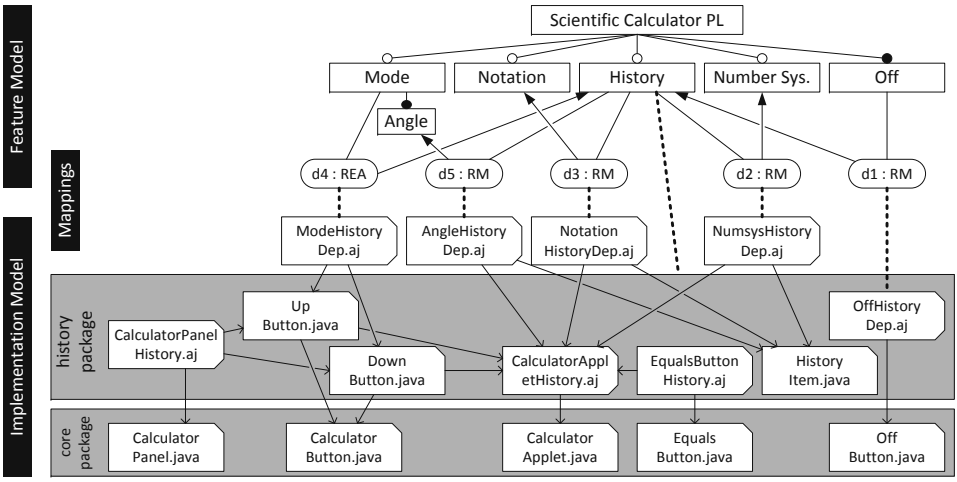

Abbildung 3: Feature-Implementation Mappings.

The feature configuration is used in *Automated Product Derivation* processes (❹ to ❻), to derive assembled executable products. The process starts with a model-to-model transformation ❹, written in ATL (Atlas Transformation Language), that creates an *Implementation Configuration* model. This model describes all components that have to be included to implement this particular feature configuration. The model is then processed by a model-to-text transformation ❺ which generates an equivalent textual configuration. Finally, AspectJ mechanisms are used to perform the assembly of the particular product ❻.

Our goal is to automate the product derivation as far as possible. Within that tool chain, only the feature configuration ❸ has to be performed *interactively*. The remaining processes (❹ to ❻) can be executed mechanically. To fully automate the approach we use Ant scripts, which orchestrate the overall process and custom Ant tasks, which integrate the various tools into this tool chain. In summary, this turns a product-specific feature configuration into an *Assembled Executable Product*.

4 Related Work

AOP techniques were originally developed to modularise crosscutting concerns. Recently, they have been used for modularising feature implementation. Godil et al. [GJ05] and Liu et al. [LBL06] applied AOP to the feature-oriented refactoring of Prevayler, an open source Java application.

Feature-oriented programming (FOP) [Bat04] is another option to implement features. FOP takes features as first-class design and implementation entities, i.e., features are designed and implemented as program refinements and composed to form a complete system.

Other techniques such as Caesar [MO04] or Framed Aspect [LR04], can be used for feature implementation. Such approaches typically support only simple feature-implementation mappings, which can lead to problems when features depend on each other. Recently, Lee et al. [LKK06] identified problems with simple mappings between features and aspects and proposed guidelines on how feature dependency information can be used for implementing features using AOP. The approach presented here differs from [LKK06] in that a systematic product derivation process is introduced.

Our approach deals with product derivation based on a feature configuration. The work in [CA05] uses templates for mapping features to different kinds of models and a model-to-model transformation to instantiate these models. However, in [CA05] the rules for filtering are described as OCL constraints, whereas we describe them as a model transformation which can partly be derived from the underlying meta-model.

The work in [VG07] is similar to our approach in that domain artifacts are expressed using models, which are then transformed. AOP is used to implement crosscutting features on code level. However, the links between features and aspects are not made explicit.

5 Conclusions

Our primary goal is to make product derivation more efficient. To this end, we (1) integrate feature modelling and AOP to structure the SPL implementation to facilitate product derivation and (2) define a model-driven product derivation process, which transforms a feature configuration into an executable product. This process is implemented as a research prototype, which automatically performs the necessary steps.

Although AspectJ has been used to demonstrate the applicability of the proposed method, more advanced techniques can be applied, e.g., Aspect-oriented architecture modelling [KTG+06]. To support software engineers in the handling of *large* SPLs we are currently extending our graphical modelling tools with improved support for the feature-implementation mappings. In addition to this we are working on techniques for reverse engineering, e.g., by extracting an AML implementation model from existing Java/AspectJ code bases. These techniques can then be used for automated analyses, e.g., to check an implementation for consistency with a specified design [JB08].

6 Acknowledgments

This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero – the Irish Software Engineering Research Centre (www.lero.ie).

Literatur

[Bat04] Don Batory. Feature-Oriented Programming and the AHEAD Tool Suite. In *ICSE '04*, Seiten 702–703, Washington, DC, USA, 2004. IEEE Computer Society.

[BNP+07] Goetz Botterweck, Daren Nestor, André Preussner, Ciarán Cawley und Steffen Thiel. Towards Supporting Feature Configuration by Interactive Visualisation. In *ViSPLE 2007, collocated with SPLC 2007*, Kyoto, Japan, September 10-14, 2007 2007.

[CA05] Krzysztof Czarnecki und Michal Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *GPCE'05*, 2005.

[CN02] Paul Clements und Linda M. Northrop. *Software Product Lines: Practices and Patterns*. The SEI series in software engineering. Addison-Wesley, Boston, 2002.

[DSB05] Sybren Deelstra, Marco Sinnema und Jan Bosch. Product derivation in software product families: a case study. *The Journal of Systems and Software*, 74:173–194, 2005.

[GJ05] Irum Godil und Hans-Arno Jacobsen. Horizontal Decomposition of Prevayler. In *CASCON 2005*, Seiten 83–100, 2005.

[HFT04] A. Hein, T. Fischer und S. Thiel. Produktlinienentwicklung für Fahrerassistenzsysteme. In G. Böckle, P. Knauber, K. Pohl und K. Schmid, Hrsg., *Software-Produktlinien: Methoden, Einführung und Praxis*, Seiten 193–205. dpunkt-Verlag, 2004.

[JB08] Mikolas Janota und Goetz Botterweck. Formal Approach to Integrating Feature and Architecture Models. In *FASE 2008*, Budapest, Hungary, 29 March - 6 April 2008.

[jsc08] Java Scientific Calculator. *http://jscicalc.sourceforge.net*, May 2008.

[KTG+06] Ivan Krechetov, Bedir Tekinerdogan, Alessandro Garcia, Christina Chavez und Uira Kulesza. Towards an Integrated Aspect-Oriented Modeling Approach for Software Architecture Design. In *AOM 06, collocated with AOSD 2006*, 2006.

[LBL06] Jia Liu, Don Batory und Christian Lengauer. Feature Oriented Refactoring of Legacy Applications. In *ICSE 2006*, Seiten 112–121, 2006.

[LKK06] Kwanwoo Lee, Kyo C. Kang und Minseong Kim. Combining Feature-Oriented Analysis and Aspect-Oriented Programming for Product Line Asset Development. In *SPLC 2006*, 2006.

[LR04] Neil Loughran und Awais Rashid. Framed Aspects: Supporting Variability and Configurability for AOP. In *ICSR 2004*, Seiten 127–140, Madrid, Spain, July 2004. Springer.

[MH03] Allessandro Maccari und Anders Heie. Managing Infinite Variability. In *Software Variability Management Workshop*, Seiten 28–34, February 2003.

[MO04] Mira Mezini und Klaus Ostermann. Variability Management with Feature-Oriented Programming and Aspects. In *SIGSOFT 2004*, Seiten 127–136, Newport Beach, CA, 2004.

[STB+04] Mirjam Steger, Christian Tischer, Birgit Boss, Andreas Müller, Oliver Pertler, Wolfgang Stolz und Stefan Ferber. Introducing PLA at Bosch Gasoline Systems: Experiences and Practices. In *SPLC 2004*, Seiten 34–50, Boston, MA, USA, 2004.

[VG07] Markus Völter und Iris Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *SPLC 2007*, Kyoto, Japan, 2007.