Information and Software Technology 55 (2013) 630–650

Contents lists available at SciVerse ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

MOD2-SCM: A model-driven product line for software configuration management systems

Thomas Buchmann *, Alexander Dotor, Bernhard Westfechtel

Universität Bayreuth, Lehrstuhl Angewandte Informatik I, Universitätsstr. 30, 95440 Bayreuth, Germany

ARTICLE INFO

Article history:
Available online 2 August 2012

Keywords:
Model-driven software engineering
Software product line engineering
Software configuration management
Feature models
Executable models
Model transformation
Code generation

ABSTRACT

Context: Software Configuration Management (SCM) is the discipline of controlling the evolution of large and complex software systems. Over the years many different SCM systems sharing similar concepts have been implemented from scratch. Since these concepts usually are hard-wired into the respective program code, reuse is hardly possible.
Objective: Our objective is to create a model-driven product line for SCM systems. By explicitly describing the different concepts using models, reuse can be performed on the modeling level. Since models are executable, the need for manual programming is eliminated. Furthermore, by providing a library of loosely coupled modules, we intend to support flexible composition of SCM systems.
Method: We developed a method and a tool set for model-driven software product line engineering which we applied to the SCM domain. For domain analysis, we applied the FORM method, resulting in a layered feature model for SCM systems. Furthermore, we developed an executable object-oriented domain model which was annotated with features from the feature model. A specific SCM system is configured by selecting features from the feature model and elements of the domain model realizing these features.
Results: Due to the orthogonality of both feature model and domain model, a very large number of SCM systems may be configured. We tested our approach by creating instances of the product line which mimic wide-spread systems such as CVS, GIT, Mercurial, and Subversion.
Conclusion: The experiences gained from this project demonstrate the feasibility of our approach to model-driven software product line engineering. Furthermore, our work advances the state of the art in the domain of SCM systems since it support the modular composition of SCM systems at the model rather than the code level.

1. Introduction

Software engineering aims at increasing the productivity of software engineers by providing powerful methods and tools for software development. Among others, software product line engineering and model-driven software engineering have emerged as complementary disciplines contributing to the achievement of this goal.

Software product line engineering [1–3] deals with the systematic development of products belonging to a common system family. Rather than developing each instance of a product line from scratch, reusable software artifacts are created such that each product may be composed from a library of components.

Model-driven software engineering [4,5] puts strong emphasis on the development of high-level models rather than on the source code. Models are not considered as documentation or as informal guidelines how to program the actual system. In contrast, models have a well-defined syntax and semantics. Moreover, model-driven software engineering aims at the development of *executable* models. Ideally, software engineers operate only on the level of models such that there is no need to inspect or edit the actual source code (if any).

Software Configuration Management (SCM) denotes the discipline of controlling the evolution of large and complex software systems. Over the years a wide variety of different tools and systems have been developed. They comprise small sized ones, like RCS [6], medium-sized systems like CVS [7] or Subversion [8] and even large-scale industrial tools such as Adele [9] and ClearCase [10].

The key function of every SCM system is *version control*, which is based on version models. While substantial differences exist between version models used in the above mentioned systems, they all share common principles like revisions, variants or state- and

* Corresponding author.
E-mail addresses: thomas.buchmann@uni-bayreuth.de (T. Buchmann), alexander.dotor@uni-bayreuth.de (A. Dotor), bernhard.westfechtel@uni-bayreuth.de (B. Westfechtel).

0950-5849/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.infsof.2012.07.010

change-based versioning. But when these version models are implicitly contained in the program code of the respective systems, reusing them to develop a new SCM system is not possible. Therefore the SCM domain is characterized by a huge amount of systems with more or less similar features, incorporating hard-wired version models which have been implemented with considerable effort from scratch.

In this paper, we present *MOD2-SCM* [11], a *mod*ular and *mod*el-driven product line for *SCM* systems primarily focusing on version control. The motivation for applying model-driven development to software product line engineering is:

Reduce effort : By combining model-driven and product line engineering, development effort is reduced significantly by automatic code generation and reuse.
Reasoning: By raising development to the model-level, reasoning about the system is easily possible.
Control: Model-driven product line engineering allows the control of couplings among the system components.
Architecture: The definition of the software architecture is possible by using models with different levels of granularity (e.g. package diagrams and class diagrams).

We applied model-driven product line engineering to the domain of software configuration management, because of:

Adaptability: As stated above, the domain is characterized by a large number of systems, which all share the lack of adaptability.
Reuse: By applying model-driven product line engineering to the SCM domain, we increase the reuse of different system components. E.g. CVS and Subversion share a lot of commonalities. Nevertheless, both systems have been developed from scratch with considerable effort.
Modularity: The project aims to develop a modular software architecture to configure SCM systems.

MOD2-SCM has been built in a Ph.D. project from 2005-2011. In addition to advancing the state of the art in building SCM systems, MOD2-SCM acts as a non-trivial case study for applying the methods and tool support developed in a companion Ph.D. project — called *MODPL* [12] — in the same period. The experiences gained from this project clearly demonstrate the feasibility and the benefits of our approach to model-driven software product line engineering. Furthermore, MOD2-SCM aims to advance the field of SCM systems by providing a library of loosely coupled components which can be combined in an orthogonal way.

This article is structured as follows: In Section 2 we briefly sketch an overview of our approach. Section 3 summarizes the key challenges which we faced in the MOD2-SCM project. Section 4 covers the analysis of the SCM domain and the corresponding feature model. The following section describes the executable and configurable domain model. Section 6 is dedicated to the evaluation of our approach. Section 7 revisits the key challenges introduced in Section 3. While related work is discussed in Sections 8, 9 concludes the paper.

2. Overview

2.1. Process

In our approach, we follow a *model-driven product line engineering process* (Fig. 1). Typically, product line engineering distinguishes between *domain* and *application engineering* [2,1]. While *domain engineering* is concerned with analyzing the domain and the development of software supporting that specific domain, *application engineering* deals with creating a specific application, i.e., an instance of the product line. In our approach, domain and application engineering differ from each other also with respect to required processes: Domain engineering requires a full-fledged *development process*, while application engineering is reduced to a simple *configuration process*. The steps of the engineering process are described below:

1. *Analyze Domain*: A *feature model* describing mandatory, optional and alternative features within the product line captures the result of the domain analysis. Typically *Feature-Oriented Domain Analysis* (*FODA*) [13] or one of its descendants (like FORM [14]) is used to analyze the domain.
2. *Develop Configurable Domain Model*: Afterwards, an executable domain model is developed which implements the features determined in the previous step. A link between the feature model and the domain model is established by annotating model elements with feature expressions.
3. *Configure Features*: In order to build a specific system with the reusable assets provided by the product line, features from the feature model have to be selected. The selected features constitute a *feature configuration*.
4. *Configure Domain Model*: According to the selection of features made in the previous step, the executable domain model is configured in an automatic process. This is done by selecting all domain model elements which are not excluded by feature expressions evaluating to false. The result of this step is an application-specific system which is executable.

2.2. Models and their relations

When developing software in a completely model-driven way, models at various levels of granularity (describing the executable domain model) exist which have to be combined with the feature model (capturing the variable and invariant parts of the product line) in order to map features to the corresponding model elements which are used to realize them.

The *feature model* [15,16] consists of a tree of features. A non-leaf feature may be decomposed in two ways. In the case of an AND decomposition, all of its child features have to be selected when the parent is selected. In contrast, for an OR decomposition exactly one child has to be selected. Depending on the respective variant of feature models, further modeling constructs are provided which refine these basic constructs [17].

The (object-oriented) *domain model* is composed of different model parts. On a coarse-grained level, *package diagrams* are used to describe the architecture of the software system. The packages are refined by *class diagrams*. Finally, *story diagrams* are used to describe the behavior of methods defined within the class diagrams. A story diagram resembles a UML2 interaction overview diagram, it realizes exactly one method of some class, and it may use classes, attributes and methods from multiple class diagrams.

A connection between the feature model and the domain model is established with the help of *annotations*. To this end, domain model elements can be annotated with features or feature expressions. Several consistency constraints have been introduced to ensure the syntactical correctness of the resulting configured domain model [18,19]. These constraints have been implemented in MODPLFeaturePlugin [20] (c.f. next subsection).

2.3. Tool chain

The MODPL approach was targeted at reusing existing and wide-spread tools as far as possible. To this end *FeaturePlugin* [21] and *Fujaba* [22] were used for feature modeling and creating