1
Introduction

Traditionally, software engineering has focused on developing one system at a time. In contrast, software product line engineering (Pohl et al. 2005, Clements and Northrop 2001) focuses on the mass development of similar but different software systems in one specific market segment. Differences between software systems are often managed in terms of features (Czarnecki and Eisenecker 2000), where one specific feature is considered as a potential configuration option (Apel and Kästner 2009). Although the resulting products are similar, each one is tailored to the specific needs of one customer. We consider a software system derived from a software product line a product. Products are described in terms of a set of feature.

Software product line engineering promises several benefits compared to one-single-system software engineering (Pohl et al. 2005, Clements and Northrop 2001). First, due to the systematic reuse of features in several products, software product line engineering offers faster production of customer-specific products with higher quality and lower costs. Second, the level of flexibility achieved by software product lines allows companies to adapt to changing markets and increment their portfolio quickly (Krueger 2001). Especially, in the domain of enterprise software systems, in which market changes frequently and the communication platforms are heterogeneous, optimized products can be fast produced to a specific environment or use case (Recker et al. 2006).

While the massive production of customer-specific products from software product lines brings many benefits, for most organizations, the associated risks, efforts, and costs might represent an adoption barrier. For example, building the initial core assets requires a nontrivial investment. Moreover, the adoption of new practices, processes, and tools also requires investment. One way to reduce investment and eliminate the many of the adoption barriers is to systematically extract the product line core assets from existing source code (Krueger 2001). The idea is to identify and distinguish features in previous developed products. Experience has shown that extract features from existing source code is an efficient way to build a software product line

(Kästner et al. 2008). First, because this practice demands more lightweight technologies and techniques that support the reuse of existing source code without much reengineering. Second, the adoption of lightweight technologies and techniques reduces the need of drastic changes on organizational structure, leading organizations to reuse their existing software, tools, process and practices.

Today, there are several means for identifying and distinguish features in existing products, ranging from code-oriented techniques (Kästner et al. 2008, pure systems 2012) to model-driven ones (Kelly and Tolvanen 2008, Voelter and Groher 2007). In practice, developers often rely on code-oriented techniques (Kästner et al. 2008). In such techniques, they identify one feature code for example annotating the source code with `#ifdef Term` and `#endif` directives, where `Term` stands for a feature. Based on a set of feature provided as a description of the desired product, developers can later derive customer-specific products using tools that exclude the code related with non-selected features.

We refer to such mechanisms as code-oriented because they directly refer to source code elements. Code-oriented techniques are attractive because they provide a simple and reusable programming model. For example, the code is annotated and removed. They are often easy to learn. The adoption is also easy. Preprocessors are already included in many languages, for example. Even modern techniques like CIDE (Kästner et al. 2008) and pure::variants (pure systems 2012) are considered as lightweight technologies that are easy to adopt in existing projects. Moreover, many code-oriented techniques are language independent and provide a uniform way of annotating the source code of different types of artifact. For example, preprocessors provide the same mechanism (i.e., `#ifdef Term` and `#endif` directives) for annotating source code elements of different granularity (e.g., classes, methods, attributes). The same mechanisms are also used for annotating different types of artifacts (e.g., C code, Java code or XML and HTML files).

Nevertheless, in some situations, specially in the engineering of enterprise software product lines, identifying feature code in existing source code might be a laborious and error-prone task. It is so because the successful development of modern enterprises software requires the convergence of multiple views. Domain experts, interface designers, database experts and developers with singular expertise and background, all take part in the process of building such a software system. Their singularities must be managed via domain-specific concepts. In this case, each participant of the development process has an individual language to solve the problems specific to its expertise. Therefore, a remarkable property of enterprise software product lines is the reuse of a range of implementation technologies in concert for addressing a series of concerns (e.g., business process, state persistency, service orchestration, graphical user interface).

Object-oriented framework (Greenfield et al. 2004, Booch 2004, Fayad and Schmidt 1997) is an example of development technology where software systems are made using domain-specific concepts as first-order concepts. One of the main advantages of object-oriented frameworks is that developers do not have to comprehend the implementation of the framework; instead, they create software systems by writing code that instantiates the framework-provided concept. The instantiation of the concepts require the developers to perform implementation choices (e.g., writing some XML code our implementing a defined interface) that are governed by the framework's programming interface.

Unfortunately, the knowledge required to identify feature code in the presence of domain concepts cannot be well captured by current code-oriented techniques. First, they do not provide views that visually discriminate the base code from feature code in terms of the used domain concepts. This is essential because code instantiating concepts often crosscut the entire product line source code and is also entangled with the code of other related concept instances. Second, code-oriented techniques are designed to operate at the level of programming language, without observing the existence of concepts and their programming interfaces. As a consequence, using the current code-oriented techniques, developers can easily introduce errors when they are identifying feature code. In general, such errors are very difficult to detect.

Despite the widespread use of code-oriented techniques, in fact, the model-driven perspective (Kelly and Tolvanen 2008, Voelter and Groher 2007) tends to be promising for identifying feature code in the context of enterprise software product lines. The model-driven techniques use domain-specific models to encapsulate the knowledge about feature code. They provide a concrete visualization of domain-specific concepts and their assignment to features, which promote improvements to the understandability of the software product line in general. The models developed can be also checked for correctness (semi-)automatically. This characteristic ensures that products always will meet a set of desired constraints. The constraints can be both rules applied over feature composition and framework's programming interface that governs the concept instantiation task.

However, the model-driven programming model might imposes several obstacles to efficiently identify features in existing source code. First, this programming model uses general-purpose transformation languages to assign one feature to its respective source code elements. This knowledge is used further to configure the source code in accordance with a set of desired features. As a drawback, it forces developers to learn intricate details of the model-driven technique. For example, as transformation languages are often based on abstract or concrete syntax manipulation (Beydeda and Book 2005), they require detailed knowledge of the metamodel where the product line is expressed. In most of cases, it would be inadvisable to force the developers to manipulate the syntax of the models. Moreover, transformations are language-specific, which make them harder to adapt and adjust to different usage scenarios. This obligates developers to create transformations that are specific for each new product line. Therefore, they do not provide a simple, uniform and easy to reuse technique, in contrast to code-oriented.

In this thesis we argue that the existing support for engineering software product line in the context of enterprise software system can be significantly improved by taking a language-oriented perspective. To obtain the benefits of the domain-specific modeling (Kelly and Tolvanen 2008, Voelter and Groher 2007) and code-oriented techniques (Kästner et al. 2008, pure systems 2012) we propose a novel technique based on domain knowledge modeling languages (DKMLs). DKML is a refinement of the concept of domain-specific languages. While domain-specific languages is a set of models that can be expressed using a collection of concepts and their mutual relations within a certain domain, DKMLs also delineate how to identify features in existing source code as domain-specific concepts.

For that, they provide means of specifying clear references across design boundaries, that is, the overlaps between source code instantiating framework-provided concepts and its conceptual definition. As a result, these languages offer concretely means of visualizing features in terms of domain concepts and consistency checking, always providing a uniform and reusable programming model. Our proposal encompasses previous domain specific-based techniques such as Language-oriented Programming (Rosenan 2010, Völter 2011, Völter and Visser 2011) and Framework-specific modeling languages (Antkiewicz and Czarnecki 2006).

We concretely present this idea in a tool, called GenArch$^+$, that has its architecture based on an universal configuration schema. This schema is essential to support the easy constructions of DKMLs. Given the well-defined semantics of DKMLs and due to the universal configuration schema, we also propose some initial editor operators that support the developers during the software product line implementation activity. Finally, we investigate the use of meta-data attached to DKMLs abstract syntax in order to provide mechanisms for automating the creation of domain knowledge models from existing source code.

We distinguish four different scenarios in which DKMLs improve visualization and support the enforcement of the programming interface provided by the frameworks. At the most basic level, the developer can simply visualize what source code elements belong to a feature as framework concepts and ensure that they are annotated accordingly. A more advanced support is visualizing and checking references between concept instances in which both must be identified as implementing compatible features. The third scenario involves searching for and validating the use of concept instances inside code customization. Finally, DKMLs support the developers reasoning about concept configuration overloading in scenarios in which the constraints that enforce concept instantiation are context sensitive.

Furthermore, we evaluate the use of DMKLs in software product line development and the exemplar languages. The evaluation is both empirical and analytical. The empirical evaluation involved measuring the influence of DKMLs on configuration knowledge comprehensibility, modularity and complexity. The evaluation showed that DKMLs in fact help to reduce the replication and verbosity when compared with traditional code-oriented techniques. The analytical evaluation focused on the validity of the languages, evaluating in what extends their application in software product line development. The results showed that in fact DKMLs improve configuration knowledge comprehension.

1.1 Contributions

We claim the following punctual research contributions of this thesis.

- The concept of DKMLs and its integration to a product line engineering process based on the extractive adoption approach.
- The identification and comprehensive description of the heterogeneous configuration knowledge problem.
- The analysis of four different scenarios in which DKMLs improve visualization and support the enforcement of the programming interface provided by the frameworks.
- Nine of the exemplar languages: Spring DKML, Jadex DKML, Struts DKML, Hibernate DKML, Spring-OSGi DKML, Spring MVC DKML.

– Empirical evaluation of the use of domain-specific modeling on the context of enterprise software product lines.

– The generic infrastructure for building software product lines using DKMLs and a set of algorithms for checking the entire software product line for consistency and reverse engineering DKMs from existing source code.

1.2 Outline of the Thesis Structure

Chapter 2: *Problem Formulation.* This Chapter discusses the problem addressed in this thesis. In particular, it illustrates some scenarios where the heterogeneous configuration knowledge problem occurs and provide some evidences that show that such a problem might be recurrent in practice. This Chapter also presents the main limitations of existing related work.

Chapter 3: *Supporting Heterogeneous Configuration Knowledge of Software Product Lines with Domain Knowledge Modeling Languages.* This Chapter presents the key ideas behind engineering enterprise software product lines with domain-knowledge modeling languages. It explains how DKMLs improve feature code visualization by putting in evidence which source code elements belong to a feature as domain concepts. Finally, it describes how framework's programming interfaces encoded in DKMLs support consistency checking and guided software product line development.

Chapter 4: *GenArch^{+}: Building Software Product Lines with Domain Knowledge Modeling Languages.* This Chapter completes the effort to improve the engineering of enterprise software product lines. It presents the proposed code-oriented technique based on DKMLs with tool support. Finally, it describes that DKMLs elements represent code patterns in the source code and that domain knowledge model can be automatically projected by attaching such patterns definition to some elements of the DKML abstract syntax.

Chapter 5: *Evaluation.* This chapter presents an evaluation of three different code-oriented techniques regarding some criteria: modularity, complexity, comprehensibility. It provides some evidences that the use of domain knowledge models reduces the number of features assignments but significantly increases the complexity of the software product line implementation. It also shows that, in general, the use of domain knowledge models does not require for developers less time to comprehend the product line. Nevertheless, it demonstrates that domain-knowledge models are useful for developers when they need to correctly comprehend the feature assignments.

Chapter 6: *Final Remarks and Future Work*. This chapter presents the final remarks and future work.