Automated Migration Support for Software Product Line Co-Evolution

Lea Gerling
University of Hildesheim, Institute of Computer Science
Hildesheim, Germany
gerling@sse.uni-hildesheim.de

ABSTRACT

The idea of automated migration support arises from the problems observed in practice and the missing solutions for software product line (SPL) co-evolution support. In practice it is common to realize new functionality via unsystematic code cloning: A product is separated from its related SPL and then modified. When a separated product and the SPL evolve over time, this is called SPL co-evolution. During this process, developers have to manually migrate, for example, features or bugfixes between the SPL and the product. Currently, there exist only partially automated solutions for this use case. The proposed approach is the first, which aims at using semantic merging to migrate arbitrary semantic units, like features or bugfixes, between a SPL and separated products.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; **Software evolution**; *Software configuration management and version control systems*; *Software maintenance tools*;

KEYWORDS

Software product line co-evolution, feature migration, evolution support, semantic merging

ACM Reference Format:
Lea Gerling. 2018. Automated Migration Support for Software Product Line Co-Evolution. In *ICSE '18 Companion: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3183440.3183441

1 INTRODUCTION

Software product line (SPL) engineering is an efficient way of developing a family of related software products via systematic reuse and the implementation of variability [3]. Nevertheless, sometimes new functionality in a SPL is realized using a *clone-and-own* approach, where products of the SPL are unsystematically cloned and modified. The advantage of this approach is, that it is fast and easy to use [6]. Due to that advantage, it is widely used in industry and the open source community as, for example, described in [4–6]. Whenever cloned products and the corresponding SPL evolve over time, we speak of *product and product line co-evolution*, or for short *SPL co-evolution* in this paper. This is not an exhaustive definition, as co-evolution can have different meanings in different contexts.

Managing SPL co-evolution is a challenging task [4]. Especially the migration of changes between the product line and its cloned products have lately become a focus of research [1, 5]. Currently, there exist no automated support approaches for the migration of, for example, features and bugfixes in SPLs. Thus, a developer has to migrate the code manually, which is a time consuming and error-prone task. The goal of my research is to close this gap by providing automated migration support for SPL co-evolution. To reach this goal, I will enhance existing merging techniques from the single system development domain with knowledge from the SPL engineering domain. The resulting merging techniques will be variability-aware and able to identify arbitrary *semantic units*.

A semantic unit consists of semantically related lines of code and is defined by their purpose. In this context, a purpose can be something like fixing a bug or adding new functionality. Furthermore, a semantic unit can be a well modularized function, or a large number of scattered lines of code. The purpose has to be defined by the user, while the identification of the semantically related implementation is performed automatically.

The focus of my research are SPLs realized via annotative mechanisms like C-preprocessor directives. Furthermore, it is necessary to differentiate between the *update* and the *feedback* scenario [2]. In the update scenario, a change made in the SPL is migrated to the cloned product. In the feedback scenario, a change made in the cloned product is migrated back to the SPL. The latter is particularly challenging, because the changes have to be enriched with variability to fit the SPL. This leads to the following research questions:

RQ1: How can semantic units and their variability be identified within the co-evolution process?
RQ2: How can the identified semantic units be migrated from a SPL to a corresponding product via software merging?
RQ3: How can the identified semantic units be migrated from a product to a corresponding SPL via software merging and with the correct addition of variability in the SPL?

2 SEMANTIC UNIT IDENTIFICATION

The identification of semantic units is part of **RQ1**. The idea is to help developers migrate, for example, a bugfix from a SPL to its related, but separated product without implementing it twice. To migrate this bugfix, it is first necessary to identify related implementations. For example, the bugfix could need a special method to work properly or it is bound to a specific product variant. To solve that problem, different information is needed, which is connected to

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3183441

the middle arrows in Figure 1. For the identification of a **Semantic Unit** the following input information is needed:

- Changes (like commits) that introduce the **Semantic Unit**
- **Variability Information** of the **Semantic Unit**
- Changed code

To get this information, a semantic unit identificator analyzes the commit history since the **Product Line$_{t1}$** and a **Product$_{t1}$** were separated. The developer has to choose which product is considered and if he or she desires to make an **Update** or a **Feedback**. A commit contains (on a line basis) the Changes, like additions or removals, made to the code. The semantic unit identificator assigns these Changes to **Semantic Units**. One line with a Change can belong to one or more **Semantic Units**.

Then the **Variability Information** of each **Semantic Unit** needs to be identified. **Variability Information** means, for example, if the **Semantic Unit** is optional for some products or mandatory for all. For that purpose, the surrounding variability implementation of the **Semantic Unit** will be analyzed. The output of the semantic unit identification process is a list of **Semantic Units**. This list contains the implementation of the **Semantic Unit**, its **Variability Information** and further descriptions, like the commit comments. Summarizing,the expected contribution of this approach is to allow automated identification of arbitrary **Semantic Units** and their **Variability Information** during SPL co-evolution.

3 UPDATE MERGING

The update scenario is the target of **RQ2** and illustrated in the right half of Figure 1. The idea is to automatically merge identified **Semantic Units** from the **Product Line** to a selected **Product** without the need of manual adaption or customization. Besides the identification of related implementations for the **Semantic Unit**, it is necessary to have information about the **Merge Target**. In summary, the following input information is needed to perform an update merge:

- **Semantic Unit** that has to be merged
- **Merge Target** information

The **Semantic Unit** and the **Merge Target** are selected by the developer. As the **Merge Target** is a **Product** and thus contains no variability, no further **Variability Information** is needed. Nevertheless, the developer could have choosen a **Semantic Unit** that is not applicable to that specific **Product**, because, for example, the affected code of a bugfix is never part of that **Product**. This problem shall be identified during the merge process. For that purpose it is necessary to combine textual, syntactic, and semantic merging techniques as well as a *three-way-merge*. A three-way-merge compares the target, the changes, and the common ancestor of both. The expected contribution of this approach is to allow automated semantic merging of arbitrary **Semantic Units** into **Products**.

4 FEEDBACK MERGING

The goal of **RQ3** is the solution of the feedback scenario, which is illustrated in the left half of Figure 1. The idea is to automatically merge identified **Semantic Units** from a **Product** to the related **Product Line** without the need of manual adaption. It will even be possible to merge all **Semantic Units** of the **Product** and, thus,

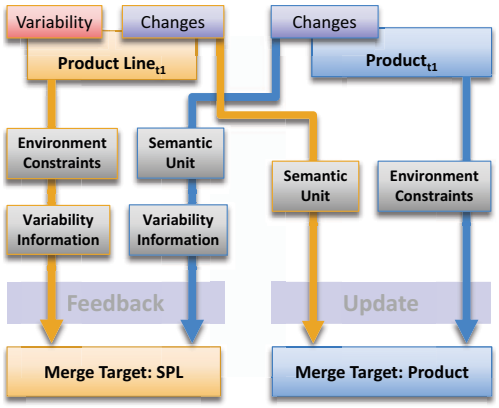

Figure 1: Automated Migration Approach

the **Product** itself back into the SPL. Besides the identification of related implementations for the **Semantic Unit**, it is necessary to have information about the **Merge Target**, its **Variability Information**, and the desired variability of the **Semantic Unit**. Summarized, the following input information is needed to perform a feedback merge:

- **Semantic Unit** that has to be merged
- **Variability Information** of the **Semantic Unit**
- **Merge Target** information

The **Semantic Unit** is selected by the developer. The origin of the **Semantic Unit** is a **Product** and therefore it contains no variability implementation. Thus, the developer has to decide whether the **Semantic Unit** will be a common part of the **Target SPL** or specific to some of its variants. This additionally needed **Variability Information** of the **Semantic Unit** has to be added during the merge process. A mutation-based approach will be used for that purpose. It can expand the existing variability implementation based on the selected desired variability and the **Variability Information** description of the **Product Line**. The **Semantic Unit** migration is then conducted as described in Section 3 with the help of textual, syntactic, and semantic merging techniques. The expected contribution of this approach is to allow automated semantic merging of arbitrary **Semantic Units** into a SPL with the correct addition of **Variability Information** via code mutation.

REFERENCES

[1] Robert Hellebrand, Michael Schulze, and Uwe Ryssel. 2017. Reverse engineering challenges of the feedback scenario in co-evolving product lines. In *21st International Systems and Software Product Line Conference*, Vol. B. 53–56.
[2] Charles W. Krueger. 2003. Towards a taxonomy for software product lines. In *5th International Workshop on Software Product-Family Engineering*. 323–331.
[3] Frank van der Linden, Klaus Schmid, and Eelco Rommes. 2007. *Software product lines in action: the best industrial practice in product line engineering.* Springer.
[4] Julia Rubin and Marsha Chechik. 2013. A framework for managing cloned product variants. In *2013 International Conference on Software Engineering*. 1233–1236.
[5] Sandro Schulze, Michael Schulze, Uwe Ryssel, and Christoph Seidl. 2016. Aligning Coevolving Artifacts Between Software Product Lines and Products. In *10th International Workshop on Variability Modelling of Software-intensive Systems*. 9–16.
[6] Ştefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. 2015. Forked and integrated variants in an open-source firmware project. In *31st International Conference on Software Maintenance and Evolution*. 151–160.