Consistent Feature-Model Driven Software Product Line Evolution

Dr.-Ing. Michael Nieke, Thesis Published: 2021-03-07

Summary. Software Product Lines (SPLs) are an approach to manage variable families of closely related software systems in terms of configurable functionality, such as the control software of cars. A *feature model* captures common and variable functionalities of an SPL on a conceptual level in terms of *features*. Reusable artifacts, such as code, documentation, or tests are related to features using a *feature-artifact mapping*. A *product* or *variant* of an SPL can be derived by selecting features in a *configuration* which is used in combination with the feature-artifact mapping to collect a set of reusable artifacts. With an automatic generation mechanism, the set of reusable artifacts is composed to a product.

Over the course of time, SPLs and their artifacts are subject to change. As SPLs are particularly complex, as they represent hundreds of thousands or more software systems at once, their evolution is a challenging task. To avoid errors introduced by ad-hoc changes and the subsequent monetary loss, SPL evolution must be thoroughly planned in advance. However, changing circumstances and requirements lead to plans not turning out as expected and, thus, replanning is required. Feature models lean themselves for driving SPL evolution as they serve as a main communication artifact and represent functionality on an abstract level without technical details. However, radical changes to feature-model evolution plans can lead to to inconsistencies as the basis for already planned subsequent evolution steps changes. These inconsistencies render the planned evolution useless. Moreover, feature-model *anomalies* that are an indicator for errors may be introduced during evolution. Such anomalies may also be source for unnecessary variability complexity, thus, leading to additional maintenance effort. Consequently, relevant configurations may come short during quality assurance as superfluous effort is spent on obsolete feature combinations, e.g., configurations of a car that can never be produced. Current feature modeling techniques are not able to ensure feature-model consistency in presence of replanning, and feature-model anomalies that have been introduced during evolution cannot be fixed efficiently.

Along with feature-model evolution, other SPL artifacts need to consistently evolve; especially for configurations, e.g., if cars are provided with over-the-air updates. Changes to an SPL may then result in undesired behavior of a product. Thus, it is crucial to understand which product(s) (configurations) are affected in which way by an update. As different engineer roles that are responsible for performing SPL evolution (domain engineers) and maintaining configurations (application engineers) typically cannot communicate with each other, product behavior may even change unnoticed. As a result, products with changed behavior may be deployed which may lead to failures and, consequently, to additional costs, e.g., if products must be recalled.

The work of this thesis provides remedy to the aforementioned challenges by presenting an approach for consistently planning and performing SPL evolution. The main contributions of this thesis can be distinguished into three key areas: planning and replanning feature-model evolution, analyzing feature-model evolution, and consistent SPL artifact evolution. As a starting point for SPL evolution, we introduce *Temporal Feature Models (TFMs)* that allow capturing the entire evolution timeline of a feature model in one artifact, i.e., past history, present changes, and planned evolution steps. Furthermore, temporal relations between the changes are modeled as first-class entities such that replanning and introduction of intermediate evolution steps is possible. In our evaluation, we show that TFMs fulfill crucial requirements for modeling evolution timelines of feature models and we used TFMs to model the evolution history of two industrial large-scale feature models. In addition to the additional benefits to enable (re-)planning, the evaluation showed that TFMs are significantly smaller than the sum of the individual feature-model versions.

We provide an execution semantics of feature-model evolution operations that guarantees consistency of feature-model evolution timelines. To this end, we formalize feature-model evolution timelines and feature-model evolution operations in terms of structural operational semantics. When new evolution operations are performed, all subsequent evolution steps are checked for violations of the formalized evolution operations. If an evolution operation would introduce an inconsistency, the reason for that inconsistency is presented and engineers cannot execute that evolution operations. This enables engineers to replan and introduce intermediate evolution operations without running into inconsistencies in later points in time. In our evaluation, we have shown that inconsistencies arising during feature-model replanning is an actual problem in industry and that our method scales for large-scale real-world feature models.

To reduce decay of feature models and highlight sources for potential errors, we introduce analyses to detect anomalies in feature-model evolution timelines. As it is pivotal for engineers to be able to fix an anomaly, we provide them with the evolution step in which that anomaly arose and with and explanation of its cause. In contrast to existing explanation methods, only highlighting structural properties of a feature model leading to an anomaly, we additionally identify the evolution operations the engineers performed which caused that anomaly. This information makes it easier for engineers to relate the anomalies to their actions. In our evaluation, we have shown that we correctly identify anomalies in evolution timelines and their causing evolution operations. Furthermore, the explanations in terms of evolution operations are up to 95% shorter than state of the art explanations of other tools.

To update products in the field after SPL evolution, we provide a methodology that enables domain engineers to reason which product configurations are affected in which way by an SPL update. Equipped with this knowledge, our methodology enables domain engineers to define automatically applicable update operations for configurations. The information provided by domain engineers then guides application engineers through configuration evolution, ensuring that no product behavior is changed unnoticed. If product behavior is changed, the engineers *know* it and can initiate appropriate actions, e.g., testing products of the respective configurations. In our evaluation using real-world SPLs, we have shown that our method provides additional benefit for updating for up to 81.2% of an SPL's configuration and it additionally detects product behavior changes for up 55.3% of an SPL's configuration which would not have been detected using other state of the art methods.

To enable easy application of our concepts, we implemented the open-source tool suite DarwinSPL[1] using a model-driven engineering approach. DarwinSPL provides user-friendly interfaces which hide the technical intricacies of our methods from end users. With the integration of all the aforementioned concepts and methods, DarwinSPL can be used to plan SPL evolution using a TFM. Its integrated and fully-automated analyses ensure that no inconsistencies are introduced and that developers are informed of anomalies and their explanations. With DarwinSPL, domain engineers can define update operations for configurations and application engineers can apply them. The concepts and the implementation in DarwinSPL are evaluated using publicly available case studies.

[1]https://gitlab.com/DarwinSPL/DarwinSPL