Supporting Feature Model Evolution by Lifting Code-Level Dependencies: A Research Preview

Daniel Hinterreiter[1(✉)], Kevin Feichtinger[1], Lukas Linsbauer[1], Herbert Prähofer[2], and Paul Grünbacher[1]

[1] Institute Software Systems Engineering,
Christian Doppler Laboratory MEVSS, Johannes Kepler University, Linz, Austria
{daniel.hinterreiter,kevin.feichtinger,lukas.linsbauer, paul.grunbacher}@jku.at

[2] Institute System Software,
Christian Doppler Laboratory MEVSS, Johannes Kepler University, Linz, Austria
herbert.prahofer@jku.at

Abstract. [**Context and Motivation**] Organizations pursuing software product line engineering often use feature models to define the commonalities and variability of software-intensive systems. Frequently, requirements-level features are mapped to development artifacts to ensure traceability and to facilitate the automated generation of downstream artifacts. [**Question/Problem**] Due to the continuous evolution of product lines and the complexity of the artifact dependencies, it is challenging to keep feature models consistent with their underlying implementation. [**Principal Ideas/Results**] In this paper, we outline an approach combining feature-to-artifact mappings and artifact dependency analysis to inform domain engineers about possible inconsistencies. In particular, our approach uses static code analysis and a variation control system to lift complex code-level dependencies to feature models. [**Contributions**] We demonstrate the feasibility of our approach using a Pick-and-Place Unit system and outline our further research plans.

Keywords: Product lines · Variation control system · Static analysis

1 Introduction

Feature models are widely used in software product lines and feature-oriented development approaches to define the commonalities and variability of software-intensive systems [1]. Frequently, features are defined for different spaces and at different levels [2,13]: problem space features generally refer to systems' specifications and are defined during domain analysis and requirements engineering; solution space features refer to the concrete implementation of systems created during development. Many techniques exist in software product lines and requirements engineering for mapping features to their implementation [1,2,5,7,16].

Such mappings are also the basis for deriving products in a feature-based configuration process to compose valid product variants automatically.

Real-world product lines evolve continuously and engineers thus need to extend and adapt feature models to reflect the changes. However, engineers require deep knowledge about the domain and the implementation to avoid inconsistencies between a feature model and its implementation [4,16]. Ensuring consistency is challenging due to the complexity of both feature-to-artifact mappings and implementation-level artifact dependencies. Checking and resolving inconsistencies is particularly important when adding or changing features during product line evolution [3].

We report our ongoing research towards an approach for lifting code-level dependencies to the level of features, thus facilitating the detection and resolution of inconsistencies. Our research is part of a project on developing a platform for distributed and feature-based clone-and-own engineering [8]. Specifically, our approach integrates feature modelling, feature-to-artifact mappings [10], and static analysis [6] (Sects. 2 and 3). It uses a revision-aware feature model [14] to track the evolution of feature models and their feature-to-artifact mappings. It further relies on static analysis for determining code dependencies. We present the results of a preliminary evaluation we conducted using the Pick-and-Place-Unit case study (Sect. 4) and provide an outlook on future research (Sect. 5).

2 Approach

Figure 1 provides an overview of our approach:

(1) The bottom layer represents the different solution space *artifacts* such as source code, models, or documents. The artifacts are managed in a tree structure. The nodes of the tree represent elements of artifacts, e.g., individual code statements or paragraphs in a document.

(2a) The approach relies on *feature-to-artifact mappings*, i.e., each artifact element needs to know to which feature it belongs. We assume that these mappings are automatically created using a variation control system (VCS) [10,15]. A VCS creates and manages mappings between artifacts and their corresponding features during development and enables the composition of different product variants using feature-based configuration. For instance, Linsbauer et al. describe how feature-to-artifact mappings are determined and kept up-to-date in the VCS ECCO [11,12]: as soon as a developer commits a new product revision or variant ECCO analyzes the changes in the features and artifacts, which then allows to incrementally add and refine the mappings.

(2b) Our approach further allows computing the complex dependencies between implementation artifacts in the *artifact dependency graph* (ADG). We realize the ADG as a system dependence graph (SDG) [9] globally representing the control-flow and data-flow dependencies in a system.

(3) As explained, our aim is to lift implementation-level dependencies to the level of features, which can then be proposed to a modeller in the *feature model* as suggestions to evolve the model. Thus, our approach combines the information

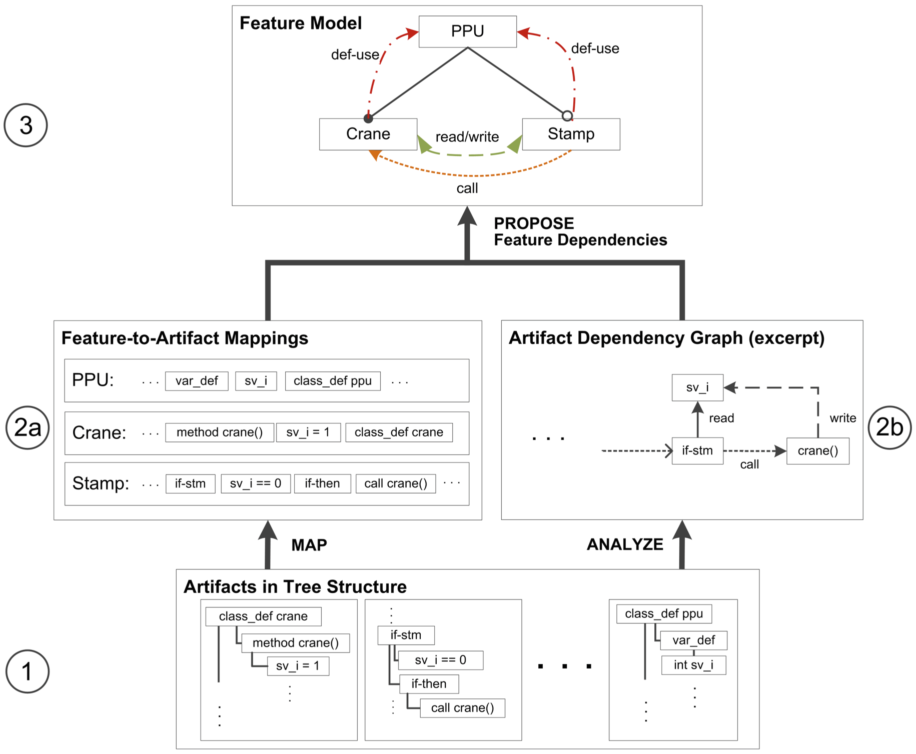
Feature Model
PPU
def-use
def-use
Crane
read/write
Stamp
call
3
PROPOSE
Feature Dependencies
Feature-to-Artifact Mappings
PPU: ... var_def sv_i class_def ppu ...
Crane: ... method crane() sv_i = 1 class_def crane
Stamp: ... if-stm sv_i == 0 if-then call crane() ...
2a
Artifact Dependency Graph (excerpt)
sv_i
read
write
if-stm
call
crane()
2b
MAP
ANALYZE
Artifacts in Tree Structure
class_def crane
method crane()
sv_i = 1
if-stm
sv_i == 0
if-then
call crane()
class_def ppu
var_def
int sv_i
1

Fig. 1. The artifact dependency graph and feature-to-artifact mappings allow proposing feature dependencies in the feature model.

from the feature-to-artifact mappings and the dependency graph. In particular, we use the artifact mappings to collect the corresponding subset of ADG nodes, which are then the starting point for traversing the dependency graph to find potential dependencies to other features. During this step, we check if we can find an artifact mapped to another feature. If so, we suggest a relation between two features to the modeller.

We distinguish between two levels of feature relations (cf. Table 1): *Dependencies* are relations required to correctly compose products. For instance, in case of a def-use dependency, i.e., one feature uses a variable or procedure declared in the implementation of another feature, a *requires* constraint must exist at the level of features to ensure that the automatically composed product compiles successfully. *Interactions* indicate weaker relations between features. This is the case, for instance, if two features write to the same variable or if one feature writes and the other reads that variable. If such interactions are not considered in the feature model, the product may still be composed, but harmful interactions may occur during execution, e.g., if two optional features write to the same output variable. This could be avoided by modeling the two optional features as alternative features.

3 Implementation

We implemented the approach by integrating a feature modeling environment with a VCS and tools for analyzing artifact dependencies. We demonstrate the feasibility of our approach by using static code analysis techniques to lift complex code dependencies.

Specifically, we adopt the VCS ECCO [11] as part of developing our feature-oriented platform. While existing VCS are mostly bound to specific artifact types [10], ECCO can be extended with plug-ins to support different domain-specific implementation languages and artifact types, as long as they can be represented in a tree structure. For example, our prototype supports source code of textual and visual languages of the IEC 61131-3 standard, Java source code, as well as configuration files for describing mappings of software variables to hardware endpoints. ECCO then creates and maintains feature-to-artifact mappings by computing differences in features and artifacts of products [12]. We do not assume initial feature-to-artifact mappings, as they can be re-created by replaying the evolution history of a system, which we showed in our preliminary evaluation. We use a system dependency graph (SDG) [6] to analyze different

Table 1. Dependencies and interactions derived from a system dependency graph.

	Type	Description
Deps	call	A feature calls a function or method of a second feature
	def-use	A feature defines a variable or constant used by a second feature
Interactions	call-call	Two features call the same function or method of a third feature
	write-write	Two features write to a data object defined by a third feature
	write-read	A feature uses data written by a second feature, while a third feature defines the data object
	read-read	Two features read a data object defined by a third feature

Table 2. Dependencies and interactions discovered for different versions of the PPU.

	PPU_v3	PPU_v4	PPU_v5
def-use	3	4	5
call	0	0	0
call-call	2	4	4
read-read	0	6	6
write-read	1	19	22
write-write	4	34	58

types of code-level dependencies (cf. Table 1) and then lift them to the level of feature models by utilizing the feature-to-artifact mappings of the VCS.

4 Preliminary Evaluation

For the evaluation of our approach we re-played the evolution history of the Pick-and-Place Unit (PPU) product line [17], thereby automatically computing the feature-to-artifact mappings using ECCO. We then analyzed feature dependencies and interactions for different PPU versions to demonstrate the feasibility of our approach. The PPU is a well-known example of a manufacturing system for transporting and sorting different work pieces. A developer of our lab (not an author of this paper) implemented different revisions and variants of the PPU using an IEC-61131-3 compliant programming language for the control part and Java for the visualization part of the system [8].

For instance, the basic version of the PPU comprises the features *Stack*, *Crane*, and *Ramp*, while the additional features *Stamp* and *Sorter* were later added to the system. As explained above, a feature model would typically become inconsistent with its implementation after such code-level changes. To show the usefulness of our support for lifting dependencies we computed the number of different types of code-level dependencies and interactions for different versions of the PPU (cf. Table 2).

We manually inspected the code with the developer of the PPU system to confirm the validity of the computed dependencies and interactions. For instance, the newly found dependencies between versions of the PPU are directly related to the addition of new features. In PPU_v3 the feature *StackCylinder* uses the variable `di_machineStarted` to check if the machine is currently running. In PPU_v4 feature a *Crane* is introduced, which also uses this variable to check the state of the machine, thus leading to a new def-use dependency. Thus, a *requires* constraint between the features *StackCylinder* and *Crane* could be suggested to the developer. PPU_v5 introduced the feature *Ramp*, leading to interactions with the feature *Crane*. Both features read and write the variable `state_crane_cur` resulting in write-read and write-write interactions showing the close relationship between these features. Although no direct constraints can be derived from such interactions, they provide highly valuable hints to developers during evolution.

Overall, the preliminary evaluation with the PPU developer confirmed most of the found dependencies and interactions.

5 Conclusion and Research Outlook

We proposed an approach that uses feature-to-artifact mappings and an artifact dependency graph to lift artifact-level dependencies to feature models. To demonstrate usefulness and feasibility of our approach we presented the number of dependencies and interactions computed for different versions and variants of the PPU case study system.

In the short term we will use the information about artifact dependencies and interactions to analyze the coupling and cohesion of features, thus supporting engineers deciding about merging or splitting features during product line evolution. This will be particularly challenging in our context of distributed feature-oriented platform evolution [8]. We will extend our dependency analysis to other types of artifacts. Our long-term plan is to evaluate our approach using large-scale product lines from our industry partner based on our earlier case studies on program analysis of industrial automation systems [6].

Acknowledgements. The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and KEBA AG, Austria is gratefully acknowledged.

References

1. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines: Concepts and Implementation. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37521-7
2. Berger, T., et al.: What is a feature? A qualitative study of features in industrial software product lines. In: Proceedings of the 19th SPLC, pp. 16–25 (2015)
3. Bürdek, J., Kehrer, T., Lochau, M., Reuling, D., Kelter, U., Schürr, A.: Reasoning about product-line evolution using complex feature model differences. Autom. Softw. Eng. **23**(4), 687–733 (2016)
4. Dintzner, N., van Deursen, A., Pinzger, M.: FEVER: an approach to analyze feature-oriented changes and artefact co-evolution in highly configurable systems. Empir. Softw. Eng. **23**(2), 905–952 (2018)
5. Egyed, A., Graf, F., Grünbacher, P.: Effort and quality of recovering requirements-to-code traces: two exploratory experiments. In: Proceedings of the 18th IEEE International Requirements Engineering Conference, Sydney, Australia, pp. 221–230 (2010)
6. Grimmer, A., Angerer, F., Prähofer, H., Grünbacher, P.: Supporting program analysis for non-mainstream languages: experiences and lessons learned. In: Proceedings of the 23rd SANER Conference, pp. 460–469 (2016)
7. Hajri, I., Goknil, A., Briand, L.C., Stephany, T.: Change impact analysis for evolving configuration decisions in product line use case models. J. Syst. Softw. **139**, 211–237 (2018)
8. Hinterreiter, D.: Feature-oriented evolution of automation software systems in industrial software ecosystems. In: 23rd IEEE International Conference on Emerging Technologies and Factory Automation, Torino, Italy, September 2018
9. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. SIGPLAN Not. **23**(7), 35–46 (1988)
10. Linsbauer, L., Berger, T., Grünbacher, P.: A classification of variation control systems. In: Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2017, pp. 49–62. ACM (2017)
11. Linsbauer, L., Egyed, A., Lopez-Herrejon, R.E.: A variability-aware configuration management and revision control platform. In: Proceedings of the 38th International Conference on Software Engineering (Companion), pp. 803–806 (2016)

12. Linsbauer, L., Lopez-Herrejon, R.E., Egyed, A.: Variability extraction and modeling for product variants. Softw. Syst. Model. **16**(4), 1179–1199 (2017)
13. Rabiser, D., et al.: Multi-purpose, multi-level feature modeling of large-scale industrial software systems. Softw. Syst. Model. **17**, 913–938 (2018)
14. Seidl, C., Schaefer, I., Aßmann, U.: Capturing variability in space and time with hyper feature models. In: Proceedings of the 8th International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS 2014, pp. 6:1–6:8 (2013)
15. Stănciulescu, S., Berger, T., Walkingshaw, E., Wąsowski, A.: Concepts, operations, and feasibility of a projection-based variation control system. In: Proceedings of IEEE ICSME, pp. 323–333 (2016)
16. Vierhauser, M., Grünbacher, P., Egyed, A., Rabiser, R., Heider, W.: Flexible and scalable consistency checking on product line variability models. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 63–72 (2010)
17. Vogel-Heuser, B., Legat, C., Folmer, J., Feldmann, S.: Researching evolution in industrial plant automation: scenarios and documentation of the pick and place unit. Technische Universität München, Technical report (2014)