Evolving System Families in Space and Time

Gabriela Karoline Michelon
LIT Secure and Correct Systems Lab
Institute for Software Systems Engineering
Johannes Kepler University Linz
Linz, Upper Austria, Austria
gabriela.michelon@jku.at

ABSTRACT

Managing the evolution of system families in space and time, i.e., system variants and their revisions is still an open challenge. The software product line (SPL) approach can support the management of product variants in space by reusing a common set of features. However, feature changes over time are often necessary due to adaptations and/or bug fixes, leading to different product versions. Such changes are commonly tracked in version control systems (VCSs). However, VCSs only deal with the change history of source code, and, even though their branching mechanisms allow to develop features in isolation, VCS does not allow propagating changes across variants. Variation control systems have been developed to support more fine-grained management of variants and to allow tracking of changes at the level of files or features. However, these systems are also limited regarding the types and granularity of artifacts. Also, they are cognitively very demanding with increasing numbers of revisions and variants. Furthermore, propagating specific changes over variants of a system is still a complex task that also depends on the variability-aware change impacts. Based on these existing limitations, the goal of this doctoral work is to investigate and define a flexible and unified approach to allow an easy and scalable evolution of SPLs in space and time. The expected contributions will aid the management of SPL products and support engineers to reason about the potential impact of changes during SPL evolution. To evaluate the approach, we plan to conduct case studies with real-world SPLs.

CCS CONCEPTS

• **Software and its engineering** → **Software product lines**; **Traceability**; **Software reverse engineering**; *Reusability*; **Preprocessors**.

KEYWORDS

software product lines, software evolution, feature-oriented software development, version control systems

SPLC '20, October 19–23, 2020, MONTREAL, QC, Canada

ACM ISBN 978-1-4503-7570-2/20/10…$15.00
https://doi.org/10.1145/3382026.3431252

ACM Reference Format:
Gabriela Karoline Michelon. 2020. Evolving System Families in Space and Time. In *24th ACM International Systems and Software Product Line Conference - Volume B (SPLC '20), October 19–23, 2020, MONTREAL, QC, Canada.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3382026.3431252

1 INTRODUCTION AND MOTIVATION

The increasing diversity of end-user needs leads to new requirements such as different platforms, operational systems, compilers, and new/customized features, i.e., system functionalities. Hence, companies must produce many variants of their software system to fulfill such demand [25]. Software product line (SPL) engineering is an approach often used to manage system families by systematic reuse of a common set of assets [29]. In SPL features are the building blocks used to distinguish its products, which characterize the system variability. Variability mechanisms can be implemented with language-based approaches, e.g., by feature-oriented programming, which consists of a composition-based approach that decomposes a system, ideally in one module or component per feature. The system variability can also be implemented by tool-driven mechanisms, such as version control systems (VCSs), preprocessors, and build systems [1].

VCSs have been used to manage concurrent variants of a system, i.e., products of an SPL, similar to a clone-and-own strategy, by their branching, forking, and merging capabilities [6]. With branches, the VCSs offer the management of variants as they enable to store and to identify versions of components of a software. However, each variant of a system is continuously maintained and evolved over time, which leads to numerous revisions of the variant [35]. For instance, a revision of a variant can be the result of refactoring, fixing a bug, improving a non-functional property, or adapting the system to a new platform or environment. These revisions can lead to many changes in one or more features or the common base code. Thus, a modification of artifacts can involve the propagation of changes and the need to merge these changes in multiple variants. Therefore, managing system families with a unified mechanism to address both system evolution in space and time is still an open challenge in software engineering, directly affecting the activities of developers and engineers and software quality [4].

Developing an SPL requires evolving the whole platform, which can affect many variants. Furthermore, over time the number of revisions and variants to handle increases, which implies dealing with a higher number of logical expressions. Hence, it becomes a cognitively complex task [16]. Existing mechanisms do not provide variability-aware change impact analysis. Thus, there is a need for a unified approach providing mechanisms to manage system families evolving in space and time. This approach should be

able to propagate changes in an automated way with consistency checking. Therefore, the goal of this doctoral work is to investigate mechanisms and define an approach for easing the evolution and management of system families in space and time.

In this way, we will investigate and propose solutions for the maintenance and evolution of SPLs by versioning systems at the level of features. Hence, managing versions of variants at the level of feature revisions will result in different implementations for the same features. Therefore, our proposed approach will provide a mechanism to identify the kinds of modification of features and to determine how these modifications affect existing variants containing their previous revision. In the case of different behaviors or feature interactions, a new revision must be added to the feature. When the behavior remains, the change will be permanently done in the revision of the feature, but recover the previous implementation must be possible. To check inconsistencies and valid configurations of feature revisions, our proposed approach also aims to provide a reverse engineering mechanism to retrieve a feature revision model from existing system variants. We thus will cope with integrated mechanisms in the VarCS to reflect the problem space through feature models containing feature revisions. Finally, we will conduct experiments with real-world scenarios, namely actively-developed open-source projects, to evaluate the usefulness of the proposed approach. We have already performed an empirical analysis of feature evolution. We assume that managing an SPL at the level of feature revisions may ease the maintenance and evolution tasks [24]. We also have conducted an evaluation of an existing technique and tool for locating feature revisions [24], which is an essential previous step to come up with a solution for automating the propagation of changes over variants.

We expect by this doctoral research to contribute to the state-of-the-art and practice by: (i) providing support for software system developers to determine and understand the impact of system families' evolution; (ii) empirical analyzing the need of evolving system families by dealing with feature revisions; (iii) easing the management of products of an SPL by means of feature revisions; and (iv) motivating tool developers to implement instances of our approach to managing system families evolving in space and time.

The remaining of this paper is structured as follows: Section 2 discusses existing mechanisms and their limitations for managing system families in space and time. Section 3 states clearly our research goal and describes the intended research methodology, as well as the proposed approach and its evaluation. Section 5 presents the preliminary results achieved so far. Finally, in Section 6 we show our work plan outlining the steps until the doctoral defense.

2 STATE OF THE ART

Modern VCSs can help to deal with versions (change history), but they are limited to aid the variability management of artifacts at a higher level of abstraction, such as the feature level. Yet, they do not properly support unified and integrated management of artifacts of an SPL, such as keeping traceability between variability information (e.g., feature models) and the specific type of artifact (e.g., source code). However, as systems rarely consist of a single type of artifact, it is necessary to use additional mechanisms to manage and evolve all artifacts based on individual features of an SPL [16]. Currently, VCSs are limited on providing mechanisms to deal with preprocessor directives (`#ifdefs`) to maintain (e.g., bug fixing) and to evolve system variants (e.g., adding a cross-cutting feature). There is no clear separation of concerns and preprocessors only operate on textual implementation artifacts like source code and cannot be used for models or diagrams [16]. Furthermore, annotations based on preprocessor directives can be error-prone, as they lead to subtle syntax errors (e.g., when an opening bracket is annotated without its corresponding closing one) [22]. Due to these limitations, we can find in the literature the proposal of variation control systems (VarCS), which provide capabilities to integrate the management of revisions and variants of software. However, they also have limitations, such as support to specific types and granularity of implementation artifacts [16]. Furthermore, according to the survey of Linsbauer et al. [16], there is not enough evidence demonstrating their success in real-world scenarios. This lack of practical evidence may hinder or not motivate the use of VarCS. In addition, VarCSs do not consider the concern of developers to be dependent on a particular style of artifact repository assumed by these systems. Thus, an approach that offers a unified mechanism for managing system families in both space and time is still missing in the literature and needs to be further explored [4].

3 RESEARCH METHODOLOGY AND APPROACH

Aiming to find possible solutions for the challenging management of system evolution in both space and time, our research is guided by an overall research goal:

RG. ***Supporting the management of system families evolving in space and time at the level of feature revisions.***

To support the system evolution in space and time, our research methodology involves several steps. Firstly, (i) understanding how features of systems evolve in space and time, in terms of their implementation and behavior, by empirically analyzing the extent and context of feature evolution. Secondly, (ii) proposing an approach to deal with feature revisions in SPLs, by easing the management of system families evolving in space and time. Lastly (iii) conducting case studies with the proposed approach for evaluating its usefulness in practice. Next, we explain in more detail how we intend to carry out each step of our empirical analysis. We also present in detail the proposed approach, discuss important implementation aspects, and show how we intend to evaluate it.

3.1 Empirical Analysis of Feature Evolution

Based on the assumption that a specific feature at different points in time can have multiple implementations and introduce different and additional system behaviors, we have pointed out the need for managing system families at the level of feature revisions. Thus, we have been conducting an empirical analysis of the feature life cycle and experiments with system families with a set of feature revisions from a real-world scenario. The empirical analysis consists of mining how much and in what context features change over time.

3.1.1 Mining how much features change over time. The goal of this step is to investigate how much features change over time in terms

of size, complexity, and behavior. This will help to comprehend how developers implement, maintain, and evolve features over time and help us to be aware of how to improve existing mechanisms for managing SPLs in space and time. For collecting this information, we need a tool able to mine feature revisions. We have already made progress on mining information on feature revisions by developing a tool to investigate the frequency of feature changes, the scope of feature modification, and the impact of changes in feature variability and complexity of SPLs in VCSs. Our tool can analyze the life cycle of features overall commits of C/C++ preprocessor-based systems managed in a VCS. We start the mining process by cloning the system repository. To collect information from the repository, we capture all commits of all releases and preprocess every C/C++ source file to get a clean version of the annotated code from macro definitions and functions. We adopt a strategy to get the features that belong to a specific block of code. Therefore, we need to analyze every feature annotated above the specific block of code that changed and has interaction with the feature from the changed block expression.

Figure 1 shows an example of conditional blocks, which allows us to explain our strategy. External features are the ones that can be selected or deselected as a configuration option in a variant. The internal features, then, are the ones that are defined at some point in the source files. In Figure 1 the features `A` and `Y` are external while `B`, `X` and `C` are internal. If a change happens in the conditional block in line 9 (the file on the left side of Figure 1), we analyze every feature that has an impact on activating the block of code in line 8. A queue of implications is created by an extraction process of configuration constraints from code [28]. To create the queue of implications, we analyze every condition above the conditional block of line 8 and the header file included, which contains `#defines` of features. The `#defines` directives also must be considered, as the feature `X` in the enclosing conditional block. The expression of the changed conditional block contains feature `B` that is defined if feature `A` is not defined. It also contains the feature `C` that is defined in the `#include` file, if feature `Y` is selected. In this example, we create a queue of implications for feature `B` where it will contain the implication: $(\neg \texttt{A} \implies B = 10) \wedge (\texttt{A} \implies \neg B)$. In cases of having `#else`, we concatenate in the same way placing as $elsePart$: $(Condition \implies (Literal = Value)) \wedge (\neg Condition \implies elsePart)$.

The example shows that manually solving constraint satisfaction problems quickly becomes a time-consuming, complex, and error-prone task if many constraints and variables are involved in a block of `#ifdef`. When features are spread across many files, and in addition to it have the influence of many defined features inside header files and/or inside many blocks of `#ifdef`, it is infeasible to manually determine the impact of changing a block of code on other features. Feature expressions may also involve arithmetic operations and comparisons with numeric values (in the range of integers or double),. Thus, Boolean satisfiability (SAT) solvers [27] are not sufficient and constraint satisfaction problem (CSP) solvers [3, 32] are needed to find possible solutions from the programming constraints. We used the CSP Choco Solver[1]. In case that conditions are not satisfiable and do not have a solution to which features belong to which block of code, we know that a specific block of code is

[1]https://github.com/chocoteam/choco-solver

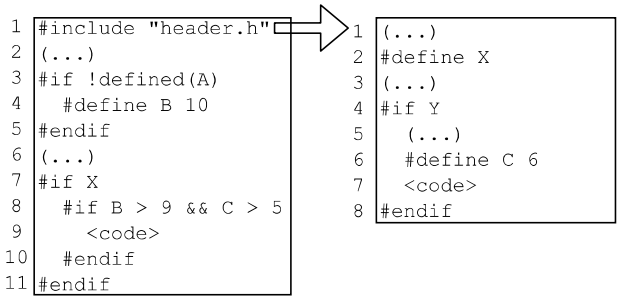

Figure 1: Example of conditional blocks.

dead code, i.e., is never executed [34]. Then, the solver receives the queue of implications built for each feature that influences on activating the changed conditional block, and we also send the expression that we would like to have a solution, i.e., to which feature a changed code belongs to. In this example, we concatenate the expression from the changed conditional block in line 8 with its enclosing parent in line 7 and with `BASE`, because in case we do not have any closest external feature to a specific changed conditional block we get a solution that the specific changed conditional block belongs to the `BASE` feature.

3.1.2 Mining what kinds of changes were made to features over time. Besides analyzing feature evolution, we want to identify at which level of granularity each change was performed and in which context the change modified the feature, i.e., refactoring or bug fix. This information is important to analyze the impact in the system behavior when using a feature revision at one point in time with revisions of features at another point in time. Hence, this information can stress the need for better mechanisms and tools for managing feature revisions. Thus, we can conduct static analysis on the feature implementation and dynamic analysis on the system behavior, before and after a change on a feature. Hence, we will be able to not only mine tangled changes at a specific point in time as well to classify features changes over time. If a feature implementation differs syntactically from one point in time to another, it may be a refactoring change to run the system faster and/or to make the code more readable. When a change is related to a bug fix on a feature, thus it may be related to a semantic difference, because a bug fix changes the feature behavior, and consequently, the system behavior [13].

Initially, to do this analysis, we will analyze commit messages to see if it is a bug fix and with static semantic analysis on the blocks of code that changed to check if it is a refactoring change. As mentioned by Herzig and Zeller [10], some commits of a system can consist of tangled changes in VCSs history, which lead to an incorrect association of changes with bug reports on commit messages. Hence, a commit can be related to changes in more than one feature, and the commit message not always reflects which features and which kind of changes were performed on them. Thus, we will also investigate some existing approaches for bug and refactoring detection to get more accurate information about features changes over time [36]. One possible attempt is to use a deep learning technique for training a neural network by using as input two chunks

of code (the code before and after the change) and as output which kind of modification it is. Ludwig[2] is an easy-to-use tool that enables us to quickly train and test deep learning models [26]. We can also rely on existing tools for clone detection, which enables us to identify the kind of change performed in a feature at a specific point in time. Clone detection approaches can be split into two categories: static analysis based approaches and dynamic analysis based approaches [14].

3.2 Approach Definition

An SPL consists not only of a concrete implementation of the system with different kinds of artifacts (known as the solution space) but also artifacts of the problem space depicting the interactions and dependencies of the system's features [1]. Therefore, our proposed approach, illustrated in Figure 2, helps to deal with the SPL engineering process in both problem and solution spaces. We split our approach into four main steps, where (1) focuses on the problem space and aims at reverse engineering a feature revision model from existing products of a system. Steps (2), (3), and (4) focus on the solution space and support the implementation and evolution of artifacts as well as the composition of products. The following sub-sections describe these steps in more detail.

3.2.1 Reverse Engineer of a Feature Revision Model (1). To help developers in deriving new variants using feature revisions, we need a mechanism to retrieve valid combinations of feature revisions. Thereby, we need to extend and adapt feature models to reflect the implementation of feature revisions. Thus, our approach will include a mechanism to reverse engineer feature revision models from existing variants of a system, according to their changes over time. The feature revision models should represent the feature sets of an SPL at many points in time. To have the information necessary to retrieve a feature revision model we will use as input (1.1) a set of existing variants and their respective configurations containing feature revisions. A revision of a feature will be a number representing that a specific feature changed, i.e., a revision will represent the feature at a certain point in time. The reverse engineering process will start by mapping the artifacts to feature revisions (1.2), which we explain in more detail in Section 3.3. Then, we will store the links showing which artifacts belong to which feature revision into a repository. The mapping will result in traces (1.3), which will be refined when committing a variant with a feature revision already linked to artifacts in the repository. To compute the feature revision model we will analyze the feature revisions' dependencies and interactions (1.4). The output feature revision model can then be retrieved (1.5).

3.2.2 Derive a new Variant (2). After using all system products to extract the existing feature revisions in the current family of systems (Step 1), we will be able to derive variants with different configurations. Hence, developers can manage an SPL evolving in space by combining different existing feature revisions. The input necessary for deriving a new variant will require a configuration with desired feature revisions (2.1) and an existing feature revision model stored in the repository (2.2). Then, the configuration will be checked if it is valid or not (2.3). In the case of a valid configuration,

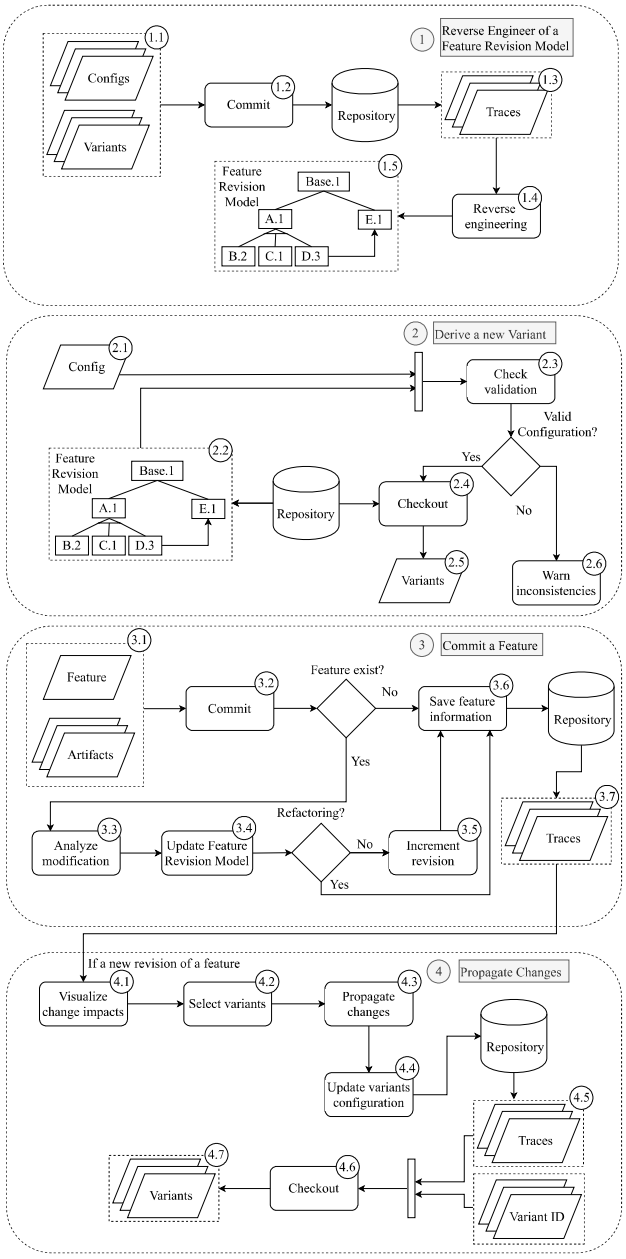

Figure 2: Workflow of the proposed approach for managing products of an SPL with a set of feature revisions.

the variant will be retrieved (2.4) as output (2.5). In the case of inconsistencies between choosing feature revisions, a warning will be raised (2.6). The warning will make developers aware that there is a missing or additional feature revision in the configuration.

3.2.3 Commit a feature (3). For evolving a variant, the approach will also contain a commit operation for an individual feature. The

[2]https://github.com/uber/ludwig