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

ACM Reference Format:
Gabriela Karoline Michelon. 2020. Evolving System Families in Space and Time. In *24th ACM International Systems and Software Product Line Conference (SPLC '20), October 19–23, 2020, MONTREAL, QC, Canada.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3382025.3414988

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: $(\neg \mathtt{A} \implies B = 10) \wedge (\mathtt{A} \implies \neg B)$. 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

```
 1 #include "header.h"
 2 (...)
 3 #if !defined(A)
 4   #define B 10
 5 #endif
 6 (...)
 7 #if X
 8   #if B > 9 && C > 5
 9     <code>
10   #endif
11 #endif
```

```
1 (...)
2 #define X
3 (...)
4 #if Y
5   (...)
6   #define C 6
7   <code>
8 #endif
```

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,

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

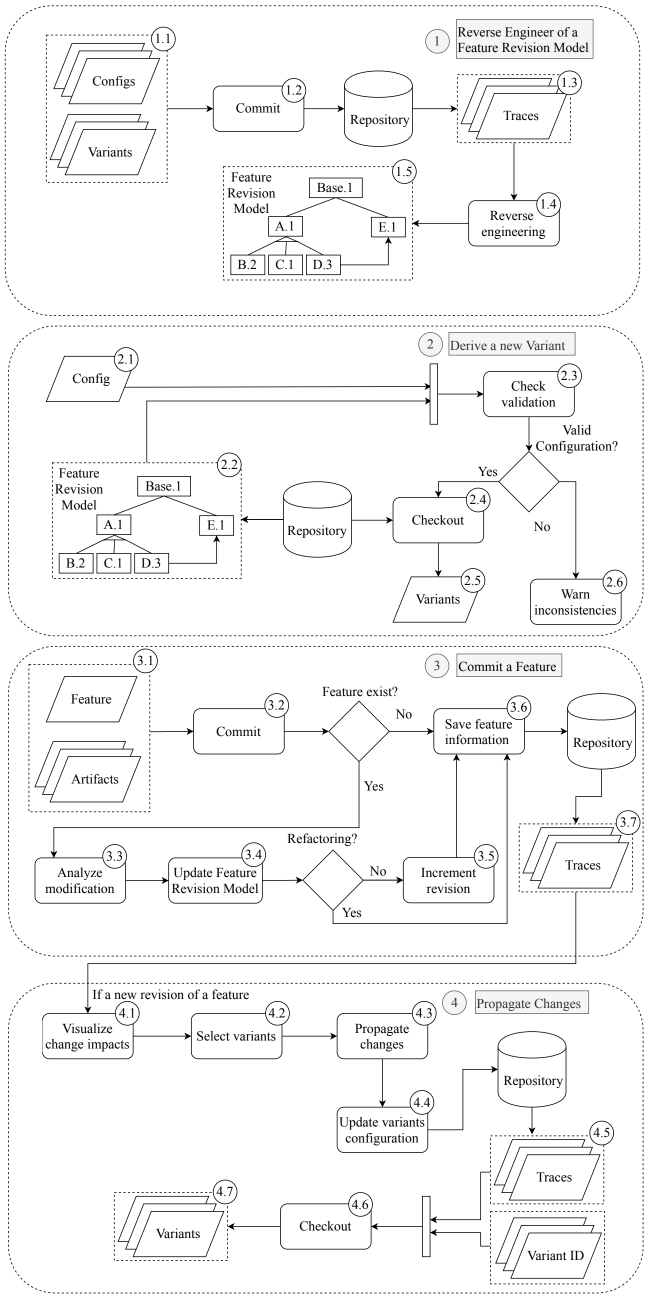

Figure 2: Workflow of the proposed approach for managing products of an SPL with a set of feature revisions.

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 input for this step will be the name of the feature and its corresponding new artifacts (3.1). Then the new artifacts will be committed (3.2) with previous analysis. If the feature already exists in the repository, i.e., it is already a feature of the SPL and possibly part of some products, a modification analysis will be performed (3.3). The modification analysis consists of verifying the change impact of the new implementation of the feature. When dependencies or interactions are added and/or removed the feature revision model will be updated (3.4). In case of a refactoring change, the feature will continue with its current revision as its behavior remains. When a feature behavior changes, e.g., due to a bug fix, its revision will be incremented (3.5). The history of the feature revision will be always stored when committing changes (3.6). This history and traces will be stored in the repository. The traces (3.7) will be used then to propagate changes over the existing products of the SPL.

3.2.4 Propagate Changes (4). When an existing feature is committed, it will contain a new revision, which means, the feature evolved over time and, consequently, some existing variants must be updated with the new implementation of the feature. Thus, developers must be able to visualize the new interactions and dependencies added and/or removed by the new revision of the feature (4.1). Next, the needed variants to be updated with the new revision of the feature can be selected (4.2) to evolve with the changes over time (4.3). Finally, the information that will be stored about existing affected variants of the SPL will be updated (4.4) with the artifacts evolved over time. Then, the repository will contain new artifacts for each variant (4.5) and can be automatically retrieved (4.6), resulting in the new revisions of variants (4.7).

3.3 Implementation Aspects

Based on the analysis of existing VarCS [16] we selected ECCO [9, 17–19] as a foundation for implementing our approach. ECCO[3] is an open-source tool and supports re-engineering variability from cloned variants by systematically and automatically reusing artifacts from existing system variants. ECCO provides feature-oriented functions to commit and checkout variants. Currently, the commit function expects the variant artifact and a configuration containing at least the feature BASE (which denotes the common artifacts of system variants) for updating the artifacts in the ECCO repository. ECCO maps feature revisions to artifacts by a mechanism implemented and shown in our work [24]. We thus will briefly explain this already implemented mechanism, which is a prerequisite for developing support for system families evolving in space and time.

Locating Feature Revisions: The approach needs two pieces of information: (i) the implementation of every existing variant; and (ii) the set of features and their respective revisions of each variant. To illustrate what are the feature revisions, we show the example in Figure 3. There are three variants as input data. The circle on the left represents a variant with feature revisions $A_1, \neg B, \neg C$. The variant represented by the circle in the middle has feature revisions A_2, B_1, C_1, and the variant represented by the circle on the right has feature revisions $\neg A, B_2, C_1$. The features A and B have two revisions, which means they do not have the exact implementation as they already had before, e.g., feature A_2 can be a new revision of the feature A due to a bug fix and/or refactoring in its artifacts. The feature location approach is based on the comparison of commonalities and differences between the artifacts of variants and their feature revisions. The artifacts of variants are organized in a hierarchical tree structure, which we refer to here as an artifact tree. An artifact can be any type of variant's implementation. In the source code, an artifact can be represented as a class node, a method node, or a single statement node in the tree structure. The commonalities and differences between the variants are analyzed by the Longest Common Subsequence (LCS) algorithm. Presence conditions are assigned for mapping artifacts of variants (as shown in Figure 3: $a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8$) and feature revisions (as shown in Figure 3: A_1, A_2, B_1, B_2, C_1). The presence conditions are computed for each artifact as a disjunctive normal form (DNF) formula.

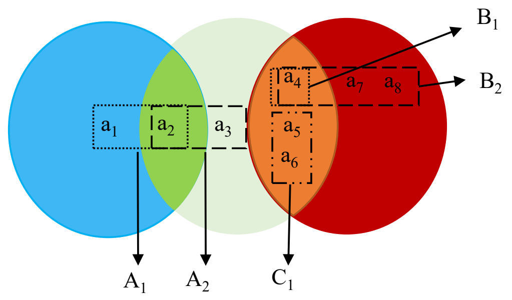

Figure 3: Artifacts of variants mapped to feature revisions.

Thus, our proposed approach will add new functionalities into ECCO to also depict the problem space of committed variants based on feature revisions. Furthermore, it will check if the configuration is valid before checking out a new variant. Committing an individual feature is also a new improvement, as ECCO currently needs as input to commit features an entire variant, with its feature revisions and BASE feature. To ease the propagation of changes over system variants we will also provide an automated mechanism to update existing variants containing a new revision of a specific feature. Next, we will explain in more detail how we plan to develop the new functionalities proposed in our approach.

Reverse Engineer of a Feature Revision Model: To implement the feature revision models, we will study a mechanism to retrieve the assets' dependencies of each of the feature revisions to find the relations between them. We also need to study how to retrieve feature revision models where their combinations are indeed well-formed. Similar to previous work from Assunção et al. [2], we plan to reverse engineer variants into feature models. However, we will use the source code of feature revisions, hence, the feature models will depict the problem space of an SPL developed with a common set of feature revisions. With feature models representing specific revisions of features, we can help to reduce the effort from configuring both dimensions of variability in space and time. Expressing the incompatibility of one version of a feature with versions of another feature, similar to the Hyper Feature models, proposed by Seidl et al. [33], must also be possible with the feature revision models.

We are just at the beginning of this step. Thus, our next task is to survey the literature on existing (temporal) feature modeling

[3]https://github.com/jku-isse/ecco.git

approaches [11]. Next, we will select a strategy for implementing feature revision models. Initially, as a first attempt, similar to Feichtinger et al. [8], we will use ECCO to compute the mappings between features and source code. We also need an approach to build the system dependence graph that contains all control and data flow dependencies of a program for finding dependencies and interactions at the level of features. Finally, to aggregate feature dependencies and interactions to suggest a feature revision model, we can use an SAT solver to compute a solution based on the formulated constraints. Thereby, by committing variants with existing feature revisions to ECCO and finding feature dependencies and interactions, we will be able to reverse engineer the variants to feature revision models.

Additional functionalities The additional functionalities (Steps 2, 3, and 4) of our approach to be implemented in ECCO requires adding a new entity model to store the history of features and variants of an SPL. To check the validation of a configuration when deriving a new variant (Step 2) we will take into account the Boolean constraints obtained with the feature revision model approach and dependencies and interactions of the feature revisions. The operation of commit an individual feature (Step 3) will use a static and dynamic analysis, which we plan to develop as mentioned in Section 3. The new added and/or removed interactions and dependencies of the feature will be synchronized with the current feature revision model which also must be stored in the repository (as an XML file). In case a feature is modified by refactoring we will just store in its history the date, developer information, and the new implementation. The feature will be linked with all its revisions. Each revision will contain information regarding the date, developer, and traces corresponding to the commit, i.e, that specific point of evolution in time. When a modification impacts on a different feature behavior we will add a new line for the new revision ID to the revision table, which is linked with the specific feature ID, and also contains revision information. The changes on a feature can be propagated over variants (Step 4) by storing information in the repository for every variant already committed and checked out. The information stored will link for each variant an ID, which also can have many revisions. The variant revision is linked with its respective feature revisions at a specific point in time. Thus, when propagating changes the link stored with an ID of each variant revision and their feature revisions will be updated with the last revision ID of the feature. Then, by the checkout operation, the traces retrieved for the variant will be updated according to the new revision of the feature. Hence, the variant composed will contain the artifacts with the last changes performed over time.

4 APPROACH EVALUATION

For evaluating the usefulness of our approach to manage system families evolving in space and time, we will experiment the new functionalities to be implemented in the ECCO tool. Thus, we need variants and their configurations, i.e., feature revisions. For this evaluation, we will use case studies with ground-truth variants generated with our mining tool. The mining tool can retrieve feature revisions from SPLs in VCSs, as illustrated in Figure 4. For the entirely proposed approach, we plan to evaluate it by conducting a case study with a realistic environment. The case study will consist

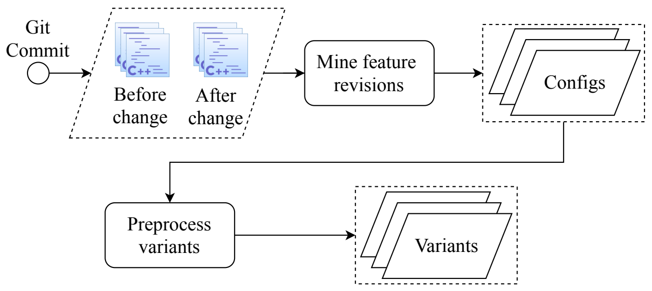

Figure 4: Process for mining ground-truth variants with a set of feature revisions for preliminary analysis of the approach.

of the use of the suggested approach implemented in ECCO by developers of an industry partner. Firstly, developers must use the tool for locating feature revisions from existing system variants. Then, developers must manipulate features individually, access the history of features, and try to propagate their changes over variants. Also, they must use the tool to derive variants with new configurations of existing feature revisions. Lastly, we will get feedback from the developers of the industry partner and conduct an empirical analysis of the usefulness of the approach in practice.

5 PRELIMINARY RESULTS

To the best of our knowledge, there are no existing studies on feature location techniques and feature models taking into account feature revisions. As mentioned by Hinterreiter et al. [11] it is essential to consider feature revisions in an SPL because features are constantly evolving. In this way, we contribute with the state of the art to mitigate this new assumption to treat revisions of features. To support this new concept, we also made improvements to our mining tool to understand how features have been evolved in SPLs overall commits in VCS. We have done an empirical analysis of the features' life cycle, which was reported in a study [24]. We also evaluated a technique for locating feature revisions, which results are also described in our work [24]. Next, we will describe briefly the achieved results.

5.1 Mining feature evolution in space and time

Our mining study of feature evolution over time in SPLs in VCS shows that a specific feature from one commit of one release can be very different from another commit of another release. This information supports the new concept of feature revision to the state-of-the-art. We mined open-source C/C++ preprocessor-based systems and computed common metrics from previous research [7, 12, 15, 20, 30, 31] to avoid varying definitions and to not limit the applicability and comparisons with our work. Based on the information mined, we have been analyzing feature evolution in space and time, i.e., when features are introduced, modified, or removed overall commits of a system. We measured the scope of feature modifications and some characteristics of features overall commits to quantify how much the features have been changed and at which level of complexity they have been implemented. As a result of the information mined, we identified for all systems analyzed, different sets of features, and many revisions across the systems' releases, representing the evolution of features both in space and time. We also analyzed some correlations of changes, which gave us an interesting empirical analysis on how this information can help to analyze tendencies of change impacts. Furthermore, as a result of this work, we provided insights that can help enhance existing VCSs to support feature-oriented development considering feature revisions. We also contribute to the availability of our dataset mined, which can be used as a basis for new studies.

5.2 Evaluating the efficiency of a technique for locating feature revisions

For preliminary experiments for evolving variants containing a set of feature revisions, we first evaluated the technique for locating feature revisions implemented in ECCO. To evaluate the technique, we needed a ground truth, which enables us to compare the retrieved artifacts from the traces located by the feature revision location technique. Furthermore, the ground truth variants cannot be at a single point in time, as we aim to evaluate our approach to managing system evolution over time. Thereby, to get variants that contain features at different points in time, we mined variants from VCSs that contain one to multiple features evolved in time, by using our mining tool developed (see Section 3.1.1). We chose open-source SPLs using the VCS Git with a considerable history of development, which has been used in previous works. By generating a considerable number of variants with feature revisions accordingly to changes on features made on VCSs, we compared the artifacts retrieved by our feature location technique with the artifacts of the ground-truth variants. The artifacts were compared at two levels of granularity: file and line. These levels of granularity have been chosen because we not only analyze C/C++ files changed but also treated binary files and other text files changed in each Git commit considered. In addition, our empirical analysis showed that features have been changed over time by adding or removing an entire file and/or fewer lines. The information retrieved from the mapping between artifacts and features in relation to the relevant information was compared by measuring precision, recall, and F1 score, which are metrics commonly used to evaluate feature location techniques [5, 21, 23]. In summary, we achieved higher precision and recall for information retrieved from the systems' variants, ranging from 99%-100% and 93%-99%, respectively, at file-level and line-level granularity. When investigating why we could not reach 100% precision and recall we found that this is due to some feature interactions, i.e., we did not consider if features are from the same `#if #else` block when combining them. Thus, as the preprocessor disregarded the `#else` block when the `#if` condition is satisfied, our ground truth variant will not contain all the features artifacts. In the case of missing artifacts in the composed variant, we have implementation limitations with the LCS algorithm. This algorithm is not perfect and can make some wrong alignments in the comparison process to refine traces. Thus, our proposed approach to reverse engineer variants into the feature revision model will allow us to analyze which feature revisions can be combined. Then, we will be able to check inconsistencies between feature revisions of a configuration and, hence, to compose valid variants.

Table 1: Schedule of the work plan for each step of this doctoral research.

Step	Status	Start	Finish
1 - Mining feature evolution	Completed	07/19	06/20
2 - Locating feature revisions	Completed	09/19	04/20
3 - Mining change impacts	In progress	07/20	12/20
4 - Implementing approach	To start	01/21	10/21
5 - Evaluating approach	To start	11/21	02/22
6 - Writing doctoral dissertation	To start	02/22	05/22

6 WORK PLAN

This doctoral research started in May 2019. We scheduled in six steps our work plan until the defense, as presented in Table 1. So far, we defined a technique for mining feature revisions in SPLs with C/C++ preprocessor directives. We used the mining technique for empirical analysis of the feature evolution (Step 1) and for evaluating the technique for locating feature revisions already implemented in ECCO (Step 2). We will conduct more empirical analysis on feature evolution to understand how the changes affect the behavior of the systems (Step 3). We expect to survey enough feature evolution information after improving our mining tool and conducting more empirical analysis. This remaining empirical work must be completed up to the end of this year. We thus plan to submit the results for a conference. After, we plan to start in January of the next year the implementation of our proposed approach (Step 4), which will consist of adding the new functionalities (see 3.3) into ECCO. The plan is to finish the proposed approach up to October 2021. Then, as the following step, we will evaluate our approach by conducting a case study with a real-world scenario (Step 5). We intend to submit the results of the Step 4 of reverse engineering feature revision models from variants to either an international journal or a conference. The complete results and empirical evaluation of the entire approach proposed, obtained from Steps 4 and 5, is planned to be reported in a journal article, focusing on qualitative and quantitative analysis of our approach. Finally, we plan to start writing the doctoral dissertation (Step 6) in February 2022, when we believe the previous steps will be all completed. We hope to finalize Step 6 in May 2022 and defend the doctoral dissertation in June 2022.

ACKNOWLEDGMENTS

I would like to thank my supervisor Alexander Egyed. I also want to kindly thank Wesley K. G. Assunção, Lukas Linsbauer and Paul Grünbacher for contributing to this work. This research is funded by the LIT Secure and Correct Systems Lab, the Austrian Science Fund (FWF), grant no. P31989; Pro2Future, a COMET K1-Centre of the Austrian Research Promotion Agency (FFG), grant no. 854184; CNPq, grant no. 408356/2018-9 and FAPPR, grant no. 51435.

REFERENCES

[1] Sven Apel, Don Batory, Christian Kstner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer Publishing Company, Incorporated, New York.
[2] Wesley Klewerton Guez Assunção. 2017. *ModelVars2SPL: an automated approach to reengineer model variants into software product lines.* Ph.D. Dissertation. Federal

University of Paraná, Curitiba, Brazil. http://hdl.handle.net/1884/47037

[3] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. 2006. *Using Java CSP Solvers in the Automated Analyses of Feature Models*. Springer Berlin Heidelberg, Berlin, Heidelberg, 399–408. https://doi.org/10.1007/11877028_16

[4] Thorsten Berger, Marsha Chechik, Timo Kehrer, and Manuel Wimmer. 2019. Software Evolution in Time and Space: Unifying Version and Variability Management (Dagstuhl Seminar 19191). *Dagstuhl Reports* 9, 5 (2019), 1–30. https://doi.org/10.4230/DagRep.9.5.1

[5] Daniel Cruz, Eduardo Figueiredo, and Jabier Martinez. 2019. A Literature Review and Comparison of Three Feature Location Techniques Using ArgoUML-SPL. In *13th International Workshop on Variability Modelling of Software-Intensive Systems* (Leuven, Belgium) *(VAMOS '19)*. ACM, New York, NY, USA, Article 16, 10 pages. https://doi.org/10.1145/3302333.3302343

[6] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *17th European Conference on Software Maintenance and Reengineering*. IEEE, San Francisco, CA, USA, 25–34. https://doi.org/10.1109/csmr.2013.13

[7] Sascha El-Sharkawy, Nozomi Yamagishi-Eichler, and Klaus Schmid. 2019. Metrics for analyzing variability and its implementation in software product lines: A systematic literature review. *Information and Software Technology* 106 (2019), 1–30. https://doi.org/10.1016/j.infsof.2018.08.015

[8] Kevin Feichtinger, Daniel Hinterreiter, Lukas Linsbauer, Herbert Prähofer, and Paul Grünbacher. 2019. Supporting feature model evolution by suggesting constraints from code-level dependency analyses. In *18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2019, Athens, Greece, October 21-22, 2019*, Ina Schaefer, Christoph Reichenbach, and Tijs van der Storm (Eds.). ACM, New York, NY, USA, 129–142. https://doi.org/10.1145/3357765.3359525

[9] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE, San Francisco, CA, USA, 391–400. https://doi.org/10.1109/ICSME.2014.61

[10] Kim Herzig and Andreas Zeller. 2013. The Impact of Tangled Code Changes. In *10th Working Conference on Mining Software Repositories (MSR '13)*. IEEE, San Francisco, CA, USA, 121–130.

[11] Daniel Hinterreiter, Michael Nieke, Lukas Linsbauer, Christoph Seidl, Herbert Prähofer, and Paul Grünbacher. 2019. Harmonized Temporal Feature Modeling to Uniformly Perform, Track, Analyze, and Replay Software Product Line Evolution. In *18th International Conference on Generative Programming: Concepts & Experiences* (Athens, Greece) *(GPCE 2019)*. ACM, New York, USA, 115–128. https://doi.org/10.1145/3357765.3359515

[12] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leβenich, Martin Becker, and Sven Apel. 2016. Preprocessor-Based Variability in Open-Source and Industrial Software Systems: An Empirical Study. *Empirical Software Engineering* 21, 2 (April 2016), 449–482.

[13] Lingxiao Jiang and Zhendong Su. 2009. Automatic Mining of Functionally Equivalent Code Fragments via Random Testing. In *18th International Symposium on Software Testing and Analysis* (Chicago, IL, USA) *(ISSTA '09)*. ACM, New York, NY, USA, 81–92. https://doi.org/10.1145/1572272.1572283

[14] Guangjie Li, Hui Liu, Yanjie Jiang, and Jiahao Jin. 2018. Test-Based Clone Detection: an Initial Try on Semantically Equivalent Methods. *IEEE Access* 6 (2018), 77643–77655. https://doi.org/10.1109/access.2018.2883699

[15] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE'10)*. ACM, New York, NY, USA, 105–114. https://doi.org/10.1145/1806799.1806819

[16] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. 2017. A Classification of Variation Control Systems. In *16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Vancouver, BC, Canada) *(GPCE 2017)*. ACM, New York, NY, USA, 49–62. https://doi.org/10.1145/3136040.3136054

[17] Lukas Linsbauer, Stefan Fischer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2015. Using Traceability for Incremental Construction and Evolution of Software Product Portfolios. In *8th IEEE/ACM International Symposium on Software and Systems Traceability, SST 2015, Florence, Italy, May 17, 2015*, Patrick Mäder and Rocco Oliveto (Eds.). IEEE, San Francisco, CA, USA, 57–60. https://doi.org/10.1109/SST.2015.16

[18] Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2013. Recovering traceability between features and code in product variants. In *17th International Software Product Line Conference, SPLC 2013, Tokyo, Japan - August 26 - 30, 2013*, Tomoji Kishi, Stan Jarzabek, and Stefania Gnesi (Eds.). ACM, New York, NY, USA, 131–140. https://doi.org/10.1145/2491627.2491630

[19] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2017. Variability extraction and modeling for product variants. *Software and Systems Modeling* 16, 4 (2017), 1179–1199. https://doi.org/10.1007/s10270-015-0512-y

[20] Kai Ludwig, Jacob Krüger, and Thomas Leich. 2019. Covert and Phantom Features in Annotations: Do They Impact Variability Analysis?. In *23rd International Systems and Software Product Line Conference - Volume A* (Paris, France) *(SPLC '19)*. ACM, New York, NY, USA, 218–230. https://doi.org/10.1145/3336294.3336296

[21] Jabier Martinez, Tewfik Ziadi, Mike Papadakis, Tegawendé F. Bissyandé, Jacques Klein, and Yves le Traon. 2018. Feature location benchmark for extractive software product line adoption research using realistic and synthetic Eclipse variants. *Information and Software Technology* 104 (2018), 46 – 59. https://doi.org/10.1016/j.infsof.2018.07.005

[22] Flávio Medeiros, Márcio Ribeiro, and Rohit Gheyi. 2013. Investigating Preprocessor-Based Syntax Errors. In *12th International Conference on Generative Programming: Concepts & Experiences* (Indianapolis, Indiana, USA) *(GPCE '13)*. ACM, New York, NY, USA, 75–84. https://doi.org/10.1145/2517208.2517221

[23] Gabriela Karoline Michelon, Lukas Linsbauer, Wesley K. G. Assunção, and Alexander Egyed. 2019. Comparison-based feature location in ArgoUML variants. In *23rd International Systems and Software Product Line Conference - Volume A* (Paris, France) *(SPLC '19)*. ACM, New York, NY, USA, 17:1–17:5. https://doi.org/10.1145/3336294.3342360

[24] Gabriela Karoline Michelon, David Obermann, Lukas Linsbauer, Wesley Klewerton G. Assunção, Paul Grünbacher, and Alexander Egyed. 2020. Locating Feature Revisions in Software Systems Evolving in Space and Time. In *24th International Systems and Software Product Line Conference* (Montréal, Canada) *(SPLC '20)*. ACM, New York, NY, USA, 12. https://doi.org/10.1145/3382025.3414954

[25] Ivan Mistrík, Matthias Galster, and Bruce R. Maxim (Eds.). 2019. *Software Engineering for Variability Intensive Systems - Foundations and Applications*. Auerbach Publications / Taylor & Francis, Boca Raton, FL, USA. https://doi.org/10.1201/9780429022067

[26] Piero Molino, Yaroslav Dudin, and Sai Sumanth Miryala. 2019. Ludwig: a type-based declarative deep learning toolbox. *CoRR* abs/1909.07930 (2019), 1–15. arXiv:1909.07930 http://arxiv.org/abs/1909.07930

[27] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. 2001. Chaff: Engineering an Efficient SAT Solver. In *38th Annual Design Automation Conference* (Las Vegas, Nevada, USA) *(DAC '01)*. ACM, New York, NY, USA, 530–535. https://doi.org/10.1145/378239.379017

[28] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. 2015. Where Do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *IEEE Transactions on Software Engineering* 41, 8 (2015), 820–841.

[29] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, Berlin.

[30] Rodrigo Queiroz, Leonardo Passos, Marco Tulio Valente, Sven Apel, and Krzysztof Czarnecki. 2014. Does Feature Scattering Follow Power-Law Distributions? An Investigation of Five Pre-Processor-Based Systems. In *6th International Workshop on Feature-Oriented Software Development* (Västerås, Sweden) *(FOSD '14)*. ACM, New York, NY, USA, 23–29.

[31] Rodrigo Queiroz, Leonardo Passos, Tulio Marco Valente, Claus Hunsen, Sven Apel, and Krzysztof Czarnecki. 2017. The shape of feature code: an analysis of twenty C-preprocessor-based systems. *Software and Systems Modeling (SoSyM)* 16 (2017), 77–96. https://doi.org/10.1007/s10270-015-0483-z

[32] Thomas Schiex and Simon de Givry (Eds.). 2019. *Principles and Practice of Constraint Programming* (Stamford, CT, USA). LNCS '19, Vol. 11802. Springer.

[33] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. 2014. Capturing Variability in Space and Time with Hyper Feature Models. In *8th International Workshop on Variability Modelling of Software-Intensive Systems* (Sophia Antipolis, France) *(VaMoS '14)*. ACM, New York, NY, USA, Article 6, 8 pages. https://doi.org/10.1145/2556624.2556625

[34] Julio Sincero, Reinhard Tartler, Daniel Lohmann, and Wolfgang Schröder-Preikschat. 2010. Efficient Extraction and Analysis of Preprocessor-Based Variability. In *9th International Conference on Generative Programming and Component Engineering* (Eindhoven, The Netherlands) *(GPCE '10)*. ACM, New York, NY, USA, 33–42. https://doi.org/10.1145/1868294.1868300

[35] Thomas Thüm, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer. 2019. Towards Efficient Analysis of Variation in Time and Space. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B* (Paris, France) *(SPLC '19)*. ACM, New York, NY, USA, 57–64. https://doi.org/10.1145/3307630.3342414

[36] Gang Zhao and Jeff Huang. 2018. DeepSim: Deep Learning Code Functional Similarity. In *26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE '18)*. ACM, New York, NY, USA, 141–151. https://doi.org/10.1145/3236024.3236068