VariantInc: Automatically Pruning and Integrating Versioned Software Variants

Sebastian Krieter
University Ulm
Ulm, Germany
sebastian.krieter@uni-ulm.de

Jacob Krüger
Eindhoven University of Technology
Eindhoven, The Netherlands
j.kruger@tue.nl

Thomas Leich
Harz University of Applied Sciences
Wernigerode, Germany
tleich@hs-harz.de

Gunter Saake
Otto-von-Guericke University Magdeburg
Magdeburg, Germany
saake@ovgu.de

ABSTRACT

Developers use version-control systems and software-hosting platforms to manage their software systems. They rely on the provided branching and forking mechanisms to implement new features, fix bugs, and develop customized system variants. A particular problem arises when forked variants are not re-integrated (i.e., merged), but kept and co-evolved as individual systems. This can cause maintenance overheads, due to change propagation and limitations in simultaneously managing variations in space (variants) and time (revisions). Thus, most organizations decide to integrate their set of variants into a single platform at some point, and several techniques have been proposed to semi-automate such an integration. However, existing techniques usually consider only a single revision of each variant and do not merge the revision histories, disregarding that not only variants (i.e., configuring the features of the system) but also revisions (i.e., checking out specific versions of the features) are important. We propose an automated technique, *VariantInc*, for analyzing, pruning, and integrating variants of a system that also merges the revision history of each variant into the resulting platform (i.e., using presence conditions). To validate VariantInc, we employed it on 160 open-source C systems of various sizes (i.e., number of forks, revisions, source code). The results show that VariantInc works as intended, and allows developers or researchers to automatically integrate variants into a platform as well as to perform software analyses.

CCS CONCEPTS

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

KEYWORDS

Version control, Forks, Variant integration, Variant-rich systems

SPLC '23, August 28-September 1, 2023, Tokyo, Japan
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0091-0/23/08.
https://doi.org/10.1145/3579027.3608984

ACM Reference Format:
Sebastian Krieter, Jacob Krüger, Thomas Leich, and Gunter Saake. 2023. VariantInc: Automatically Pruning and Integrating Versioned Software Variants. In *27th ACM International Systems and Software Product Line Conference - Volume A (SPLC '23), August 28-September 1, 2023, Tokyo, Japan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3579027.3608984

1 INTRODUCTION

Many organizations and open-source projects develop customized variants of their software systems, for example, to fulfill varying hardware specifications or user requirements. To implement such variant-rich systems, developers usually rely on cloning, a platform, or a combination of both [5, 17, 36, 39, 46, 57, 65, 69]. Especially when developers start to reuse and customize an existing system to derive a new variant, they rely on cloning; that is, creating a copy of the system and adopting that copy to changed requirements. Cloning is a cheap and readily available reuse strategy that is well supported by version-control systems like Git (i.e., branching) and software-hosting platforms like GitHub (i.e., forking) [24, 36, 43, 49, 65]. For simplicity, we refer mostly to version-control systems and forks in the remainder of this paper.

Usually, forks are intended for short-term development, for instance, to fix a bug or implement a new feature, and should be re-integrated into the main system (via pull-requests and merging). However, many developers use forking to implement separated variants that are not re-integrated [36, 65, 66], for instance, because these variants comprise highly innovative features that shall not be part of the base system at that point in time. In such situations, it can become drastically more expensive to maintain the independent variants, since changes must be propagated and developers can easily loose their understanding of what features are implemented in what variants [1, 7, 15, 17, 33, 35, 36, 46, 52, 61].

To tackle these problems, developers often re-engineer their cloned variants into a platform that integrates features into a common code base [9, 46, 52, 73]. A platform usually builds on a variability mechanism, such as the C preprocessor (CPP), that allows to control the implemented features [5, 23], variability models to define feature dependencies [14, 56, 62], as well as tools to configure and automatically derive a variant of the platform [5, 57]. Such a platform yields reduced development and maintenance costs, allows faster delivery of variants, and improves the system quality [36, 69]. However, re-engineering cloned variants is an expensive investment and may not achieve the expected benefits [11, 38, 63].

While the two strategies of cloning and platform engineering are often clearly distinguished in research, they are actually used in parallel by most developers [22, 36, 41, 43]. In practice, developers use a variability mechanism in their base system (i.e., a platform) and use forks to evolve that platform or create highly innovative variants. Consequently, integrating forks of a variant-rich system is a practically important problem. Several techniques for analyzing cloned variants [18, 73], supporting developers during the re-integration [19, 54], and the semi-automated integration of forks [49, 61, 64] have been proposed [6, 47, 66]. However, current research also indicates a limitation of such techniques: They do not allow to (automatically) integrate forks while also re-engineering the revision histories of individual features (i.e., allowing to trace the revisions of a specific feature in the integrated platform). To achieve traceability through time and space [3, 4], researchers have started to revisit variation-control systems [8, 50], but these require support for re-engineering variant-rich systems from pure version control to variation control to facilitate their adoption.

In this paper, we propose VariantInc (*Variant Incorporating*), a technique to automatically integrate forks of a variant-rich system into a platform. A particular property of VariantInc is that it *analyzes and prunes the revision histories of the forks (not only the most recent revision) to track what variations (e.g., bug fixes, features) were introduced at what point in time.* To manage not only variations of the different variants (e.g., features) but also their evolution, we add annotations to control these variations in the platform. We remark that a fully automated solution on its own is not ideal in practice, since developers still need to interact and assign features to the code variations that existed in the forks (i.e., concept assignment and feature location can only be solved by developers) [10, 49, 70, 71]. VariantInc is supposed to support developers with the following **U**se **C**ases: **(UC_1)** simplify and analyze revision histories of forks; **(UC_2)** identify variations in forks; **(UC_3)** automatically integrate forks into a platform; **(UC_4)** configure variations in space and time; and **(UC_5)** migrate towards variation control (cf. Section 2 for a more detailed motivation of each use case).

To this end, our contributions in this paper are:

- We propose VariantInc for automatically integrating forked variants and their version-control histories.
- We validate VariantInc by employing it on 160 variant-rich systems with varying properties, including systems with over 6,000 forks, 100,000 revisions, and 10 MLOC.
- We provide an open-access repository with the source code of our prototype and validation data.[1]

With VariantInc, we aim to support developers during variant integration processes. Moreover, we intend to provide a basis for future research on software evolution and re-engineering, particularly from pure version control towards variation control.

2 STATE-OF-THE-ART AND MOTIVATION

In this section, we provide the background needed to understand VariantInc. We further describe the related work, based on which we motivate the five primary use cases for VariantInc in more detail.

[1]https://doi.org/10.5281/zenodo.8048579

2.1 Background

Version-control systems and software-hosting platforms are established tools for managing the evolution of software systems [12, 43, 65]. While there are conceptional differences between individual tools, the core ideas are similar. A system is managed in a repository, which developers clone into a local copy to implement, commit, and push changes back into the repository. Each commit creates a new revision of the system, allowing developers the option to restore a specific state of that system by selecting a revision. So, version-control systems enable developers to configure a system by selecting specific revisions with their corresponding features—representing *variations in time* [3, 4, 8]. Most version-control systems provide even more advanced functionalities. In particular, developers can fork a repository to create a separated clone of their system. While forks are often used to implement a new feature and re-integrated into the original repository, they are also used to maintain separate variants of the system, for instance, for specific releases or customers [24, 36, 43, 49, 65, 74]. At this point, different forks may comprise new or customized features—representing *variations in space* [3, 4, 8].

Managing a variant-rich system based on cloning easily increases development and maintenance costs, requires developers to propagate changes between variants, and hampers the understanding of existing features and variants [7, 17, 33, 35, 36, 38, 46, 52, 61]. Consequently, most organizations at some point adopt software platforms by re-engineering the cloned variants [52, 73]. For this purpose, they adopt principles of software product-line engineering to establish their configurable platform [5, 41, 57, 69]. However, despite extensive research on re-engineering platforms form cloned variants, it remains a challenging and costly process [6, 11]. With the increasing use of version-control systems, more and more platforms are adopted by integrating previously separated forks. Such an integration is fundamentally different from simple merges in version-control systems and challenges a consistent evolution (e.g., matching revision histories of features before and after the integration) of a variant-rich system.

2.2 Related Work

Some researchers have been concerned with manual variant integration [1, 15, 25, 27, 44, 46] or the re-engineering of a single variant into a platform [13, 28, 51, 68]. Moreover, several techniques have been proposed to facilitate or automate parts of the integration of variants and particularly forks. However, existing reviews of the literature [6, 20, 47] show that most of such techniques focus on the analysis (e.g., feature identification [74] and feature location [16, 60]), with few tackling the actual integration of cloned variants into a platform. For instance, Rubin et al. [61] propose a framework that builds on seven operators that define how to re-engineer clones into a platform, but these have not been implemented. Similarly, Martinez et al. [54] and Fischer et al. [21] propose frameworks for analyzing and re-integrating cloned variants. In the same direction, we [34, 42, 64] have proposed techniques for supporting the merging of forked variants, for instance, by focusing on their test cases or providing visualizations. Other researchers, for example, Fenske et al. [19], Alves et al. [2], or Xue [72] describe concrete refactorings for integrating cloned variants. Closest to our technique is the work of Lillack et al. [49]. They implemented INCLINE to support developers during the variant integration by guiding their decisions of how to perform the integration of specific code blocks (i.e., variations) based on intentions.

In contrast to such works, we are concerned with automating the integration of variants implemented in forks, while also uncovering the revision histories of the integrated variations. None of the existing techniques we are aware of combines such an analysis of variations in *space and time*. Moreover, most existing techniques and refactorings focus on supporting specialized mechanisms, such as feature-oriented programming [58], aspect-oriented programming [31], or the CPP. With our technique, we aim to support any system that is managed in a version-control system, independent of its properties. However, due to its predominance in practice [5, 26, 48], we follow the same idea as Lillack et al. [49] to support particularly the CPP, and building on its concept of presence conditions to record what variations our technique integrates into the platform. A presence condition is an established concept for variant-rich systems that consists of a propositional formula over the features in a variability model, determining whether a variational element (i.e., a file) is present under a certain configuration (i.e., a configured variant) [5, 29, 45, 50, 55].

2.3 Goal and Use Cases

With VariantInc, we aim to facilitate the integration of co-evolving forks into a platform, which can then be maintained in the version-control or a variation-control system. Consequently, we aim to support any system that is developed in a version-control system, comprising variations in time (i.e., revisions) as well as potentially different variations in space (i.e., forks and a variability mechanism). The mixture of these different variations, for instance, a feature that was already implemented in the repository using the CPP and has been revised in multiple forks, challenges the variant integration into a platform and the merging of revision histories (cf. Section 4.2). Moreover, while manually integrating only the latest revisions has been the idea of most existing works, this does not scale with the numerous forks that exist for variant-rich systems—some comprising changes affecting the same code or even identical changes [65]. Simply merging the forks automatically does not ensure a reasonable system, since integration conflicts must be resolved manually [49], and ignores the revision histories (i.e., considering only the most recent revisions). To tackle these issues, we propose VariantInc, which supports developers by pruning and merging revision histories, integrating the existing variations in space and time, and providing presence conditions that allow to configure these variations in the resulting platform. Developers can use VariantInc to integrate selected forks, to automatically analyze and simplify revision histories in terms of variations, or to perform an automated integration and revise the resulting platform (i.e., assign features to the integrated variations). Concretely, we defined five use cases for VariantInc.

Prune and Analyze Revision Histories of Forks (UC$_1$). To integrate variations in space and time, we need to analyze, simplify, and merge the revision histories of different forks. The immediate result is an overview of when what variations have been introduced and a comparison between forks, allowing developers to understand their evolution. For instance, developers can see what lines of code from what forks belong to what existing variation, and how these changed over time, revealing potential alternative implementations in different forks.

> ⇒ *VariantInc provides a novel overview understanding of a system's evolution, offering developers a means to better analyze and compare co-evolving variants.*

Identify Variations in Forks (UC$_2$). By analyzing revision histories, VariantInc automatically identifies variations as well as their alternatives and redundancies, helping developers to understand the intention of a specific change (e.g., bug fixes, new features). For example, Zhou et al. [74] proposed a technique to identify features that have been implemented in a fork. As they showed, correctly identifying features requires a developer to decide on whether an identified variation represents an actual feature of the system. VariantInc automatically identifies variations, is not limited to features, and handles alternative as well as redundant changes, providing additional means to developers. The results can be enriched with specialized techniques (e.g., by Zhou et al.) or mapped by developers, for instance, to actual features or bugs.

> ⇒ *VariantInc provides an advanced analysis of the variations introduced among different forks to help developers decide which are relevant for a platform.*

Automatically Integrate Forks into a Platform (UC$_3$). Building on the previous two analyses, VariantInc can address our overarching goal by fully automatically integrating variations and annotating them with presence conditions that reflect their variability in space (i.e., the forks variations stem from) and time (i.e., their revisions). Since feature identification [74] and location [10, 60, 70, 71] in code cannot be fully automated [37], developers will need to review the integration, for example, to assign proper feature identifiers or assign variations to previously existing features. Still, VariantInc is a helpful means to derive a platform with presence conditions that developers can build upon.

> ⇒ *VariantInc provides the ability to fully automatically integrate variations of forks into a configurable platform, facilitating manual integration processes.*

Configure Variations in Space and Time (UC$_4$). The platform we derive with VariantInc includes presence conditions for variations in space and time, allowing developers to immediately configure the variations of specific forks and different revisions. So, developers can immediately restore any of the integrated forks and configure new combinations. Particularly, they can try to combine specific feature revisions to understand their interactions with the remaining system. This greatly facilitates the integration of variations into a single platform and the introduction of a platform.

> ⇒ *VariantInc provides a platform that developers can configure based on variations in space and time, ensuring that all existing (and potentially new) variants can be instantiated.*

Migrate towards Variation Control (UC$_5$). Variation-control systems [50] aim to manage variations in space and time simultaneously, for instance, by building on feature revisions instead of system revisions [3, 4]. Any system may be migrated towards variation control to facilitate the management and evolution of its variations. While we have not implemented this migration ourselves

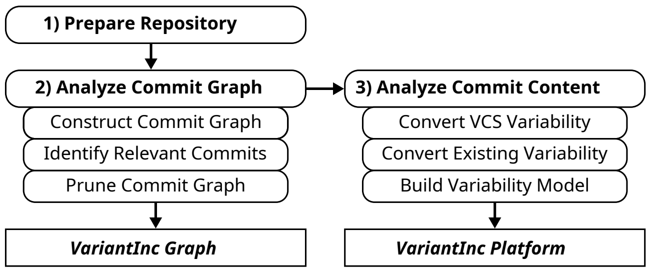

Figure 1: Overview of the process for constructing VariantInc's two core data structures, the *graph* and the *platform*.

(it would be highly tool-specific), VariantInc provides a configurable platform and all information on revision histories that are needed for such a migration.

> ⇒ *VariantInc provides the artifacts required to migrate a system and its forks towards variation control.*

3 VARIANTINC

In this section, we introduce VariantInc and its two core data structures. Moreover, we describe how each of the data structures aligns to the described use cases. We display an overview of the process for constructing VariantInc in Figure 1, and describe the details of this process in Section 4.

3.1 Overview

We aim to support developers during the mentioned use cases by extracting information from the version-control system and variability mechanism of a system into VariantInc. The automated process to build this platform consists of three major steps, *1) preparing the system repository*, *2) analyzing the commit graph*, and subsequently *3) analyzing the commit contents*. Steps 2 and 3 each yield a data structure containing information about the system. In Step 2, VariantInc produces an intermediate data structure, the *VariantInc commit graph*, which can already be used to support **(UC_1)**. Performing Step 3 yields the *VariantInc platform*, which provides the additional information to support **(UC_2)** – **(UC_5)**. In the following, we describe these two data structures and explain how they are used to support the respective use cases.

3.2 VariantInc Commit Graph

The VariantInc commit graph is a modified representation of a commit graph of a version-control system including all forks and branches. It differs from a regular commit graph in two major points. First, the VariantInc commit graph contains only commits relevant for the use cases of VariantInc—and which commits are considered relevant depends on the setup of the build process, which we explain in Section 4. Note that the content of omitted commits is not lost, as it is preserved in the computed difference between the remaining, relevant commits. Second, each commit in the VariantInc commit graph references its successors and predecessors, instead of only its predecessors. This allows for efficient backwards and forwards traversal of the revisions kept in the VariantInc commit graph.

Prune and Analyze Revision Histories of Forks (UC_1). As the VariantInc commit graph does contain relevant commits only (i.e., those representing variations), it effectively hides irrelevant information from the developer. This makes it easy to spot forks that contain actual variations. It also highlights which commits exist that are not yet integrated into other variants. These variations can then be propagated by the developer by integrating the corresponding commits into another variant. Furthermore, commits in the graph that have multiple successors indicate a common code base between two or more revisions that could be inspected further to understand what variations have been introduced.

3.3 VariantInc Platform

The VariantInc platform incorporates all forks and branches of a system in a single data structure. It contains the contents of all files from all commits in the VariantInc commit graph, including the files' histories. In addition, the platform comprises a variability model that maps all commits in the VariantInc commit graph (i.e., variations in time) and all changes of all versions of the system (i.e., variations in space). Every commit is represented by a new feature in the variability model, and their dependencies are derived from their relationships in the VariantInc commit graph. The variability model has two main purposes. First, to specify all possible variations of the variants of the system (i.e., combinations of commits and features). Second, to determine the dependencies between these variations in the platform, for example, to decide which features in which commits can be checked out together to represent a valid (i.e., error-free) configuration of a specific variant.

In more detail, the content of each file that is part of a specified variant depends on the features configured in the VariantInc platform. While configuring, each feature can be either selected or deselected, including or excluding files and their contents that do not fulfill the presence conditions of the resulting configuration. So, the VariantInc platform allows developers to immediately configure and checkout (at least) the integrated variants.

Unfortunately, similar to most version-control systems, configuring all different kinds of file types on the same granularity is hardly possible. For instance, binary files can only be present in their entirety, and thus we store their revisions similarly to Git. Consequently, while we focus on human-readable text files (i.e., source code that is configurable line-by-line), we also consider binary files and store them as single byte arrays. Every file within the VariantInc platform is uniquely identified by its path in the original version-control repository. To determine whether a file is present in a certain variation, and whether it is binary or textual, each file has a propositional presence condition.

Other variable elements in the VariantInc platforms are the actual lines of textual files and byte arrays of binary files. Every line and byte array is assigned a presence condition as well. For textual files, the presence conditions of single lines incorporate not only the history from the version-control system, but represent also the variability of any existing variability mechanism, such as the CPP. That means, a presence condition of a line is true under a given configuration only if the line is present in the configured variation *and* the configured variant. We describe how we compute presence conditions in Section 4.

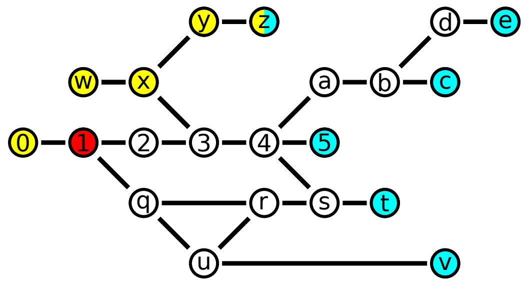

Figure 2: Initial commit graph.

Identify Variations in Forks (UC$_2$). By combining the VariantInc commit graph with the platform, we can now compare the contents of forks. Thus, developers can focus their analysis on the actually relevant variations in the source code, as VariantInc again hides irrelevant content (e.g., intermediate changes, reverted commits). Moreover, it is easier to trace changes to specific revisions and forks, helping developers to identify where a variation stems from.

Automatically Integrate Forks Into a Platform (UC$_3$). After constructing the VariantInc platform, all forks are integrated into a single platform. Developers can then review and change that platform, for example, to assign proper feature names, remove unwanted variations, or define dependencies between variations (e.g., avoiding that unintended or faulty configurations are possible).

Configure Variations in Space and Time (UC$_4$). Previously existing variations in time are for now encoded as variations in space, allowing us to unify the variability mechanism used. For instance, instead of checking out a revision of a system in a specific fork and then configuring its CPP, developers can now decide to integrate all variations into the CPP.

Migrate Towards Variation Control (UC$_5$). The VariantInc platform also provides the basis for migrating towards variation control. Specifically, instead of having revisions only for the full system, we now have variation points that align to the revisions of variations. So, by analyzing and parsing those, the platform could be migrated towards a variation-control system.

4 BUILDING VARIANTINC

To enable the automated integration of variants, we need to extract information about features and their revisions from an existing version-control system. Since we are concerned with a fully automated process for constructing VariantInc, our technique can naturally be improved by taking domain knowledge of a systems' developers into account. Therefore, we provide opportunities for developers to control the intermediate steps of the process. In the following, we explain the individual steps of that process, for which we provide an overview in Figure 1. We remark that we focus on Git and GitHub with their branching and forking strategies, respectively. However, the concepts of our process can be adopted for other version control systems and platforms with their specific mechanisms.

4.1 Prepare Repository

In the preparation step, we collect all necessary information to transform a system into a VariantInc platform. As initial input for constructing VariantInc, we require the repository of a software system. This repository must contain all relevant commits and branch labels. This also includes all commits and branch labels of any external fork deemed relevant. Furthermore, one branch must be declared as the *main branch*, which is required for integrating forks and handling orphan branches later on. Some platforms, such as GitHub, allow us to extract all of this information automatically (see Section 5 for further details). However, for other version-control systems, not all of the required information may be available. Thus, it is also possible to specify this information manually.

4.2 Analyze Commit Graph

The second major step in the automated VariantInc build process is to construct the VariantInc commit graph. We divide this step into three consecutive substeps: *1) constructing the initial commit graph*, *2) identifying relevant commits*, and *3) pruning the graph*.

4.2.1 Construct Initial Commit Graph. We start constructing the commit graph by collecting all variants of the system and their corresponding branches in the Git repository. In our automated process, we pick every branch within the repository that contains at least one commit that has not been merged into the main branch, yet. Furthermore, when analyzing software systems with forks (i.e., from GitHub), we determine all publicly available forks of the system, and analyze the corresponding main branch as well. Developers can narrow down this selection by excluding branches or forks that do not contain relevant commits (e.g., branches for bug fixing). Moreover, developers can also specify additional branches from other forks of the repository by providing an URL pointing to the fork and the tag of the branch.

From the collected branches, we extract one starting commit per branch. In the automated process, we use the most recent commit of each branch. For each of these commits, we traverse their previous commits until we find a common predecessors with the main branch (i.e., a merge base). Based on this traversal, we then build a bi-directional commit graph of all commits between the starting commits and the merge base. Due to orphan branches in Git, it is possible to have more than one root commit in one repository. Thus, it may be impossible to find a single merge base for all starting commits. In such cases, we exclude all commits that do not share at least one commit that we can reach from the most recent commit of the repository's main branch (i.e., orphan commits). In addition, we also consider all commits that are predecessors of the merge base as orphan commits, as it is not possible to reach the merge base from these commits by backwards traversal.

We illustrate our automated construction process using a small example, for which we depict the initial commit graph in Figure 2. In this example, we depict all commits of a system, with the most recent commits displayed on the right. The commits are labeled with numbers (for the main branch) and letters (for branches and forks comprising variants), and are connected to their predecessors (and consequently also their successors). We highlight all commits that are considered starting commits for the construction process in cyan (i.e., commits *5*, *c*, *e*, *t*, *v*, and *z*). As we can see, commit *5* is

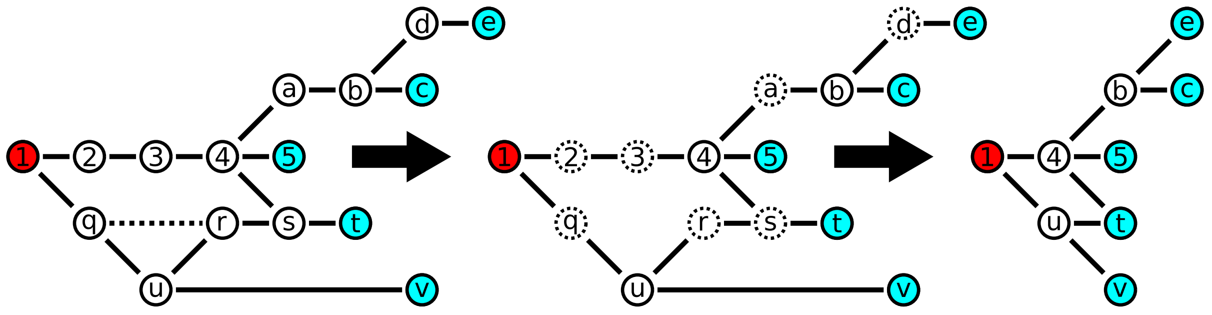

Figure 3: Pruning the commit graph from Figure 2.

the most recent commit in the main branch. Yellow commits are considered orphans, since they have no predecessors from the main branch (i.e., commits *0*, *w*, *x*, *y*, and *z*). Disregarding the orphan commits, we compute that commit *1* (highlighted in red) is the merge base (i.e., the most recent common predecessor) for the set of all starting commits.

4.2.2 Identify Relevant Commits. The initial commit graph contains every commit from the original version-control repository and also its forks. This may include commits that do not or only partly introduce variability-related changes. Thus, we remove all commits that are not considered relevant for our uses cases. In the automated building process, we decide whether a commit is relevant or not by using a simple strategy: We consider a commit as relevant, if it is not an orphan commit and is either a starting commit or has at least two successors (i.e., variations have been introduced after the commit). Users may modify this selection by manually marking further commits as relevant or irrelevant.

Naturally, we mark the starting commits as relevant, because they contain the latest variants of the system. From there, we are interested in commonalities between these variants. A commit that has two or more successors can be seen as a partial merge base for some of the starting commits, and thus probably contains a common code base. In contrast, for a commit with just one successor, there exists at least one commit (e.g., the successor), which contains more information and is either a starting commit or a partial merge base. For orphan commits, we can typically not find a merge base with other commits, which is why these also do not contribute to a common code base.

4.2.3 Prune Commit Graph. After identifying all relevant commits in the initial commit graph, we remove all non-relevant commits by pruning the graph. We begin by removing all orphan commits from the graph. As an example, we depict the resulting commit graph for Figure 2 after this first pruning in Figure 3 on the left.

Next, we repeatedly apply two steps until VariantInc reaches a fix point. In the first step, we remove certain transitive connections in the graph. For every commit, we traverse backwards through the graph and check whether one of the commit's direct predecessors can also be reached via a different path. If so, we remove the direct (i.e., transitive) connection from the commit to its predecessor. In the second step, we mark commits as not relevant according to our strategy presented earlier. Then, we remove all marked commits from the graph. When removing a commit, we keep all transitive connections between all other commits by replacing the connections of the commit's predecessors and successors. In detail, we replace all connections that point to the removed commit as a successor with connections to the commit's former successors and we replace all connections that point to the removed commit as a predecessor with connections to the commit's former predecessors. Removing a transitive connection may result in new commits with only one successor, which is considered non-relevant by our strategy. Similarly, removing a commit may lead to more direct transitive edges that can be removed. Thus, we apply both steps repeatably until the graph does not change any further.

We depict the process of pruning the commit graph in Figure 3. On the left side, we marked the direct transitive connection between commit *q* and commit *r*, as it is possible to reach *q* from *r* via *u*. In the middle, we display the graph without this connection and marked all non-relevant commits (i.e., commits *2*, *3*, *a*, *d*, *q*, *r*, and *s*). On the right side, we depict the final commit graph with all non-relevant commits removed.

4.3 Analyze Commit Content

After constructing the VariantInc commit graph, the next step is to build the VariantInc platform. Again, we divide this step into three substeps: *1) converting variability from the version-control system* into presence conditions, *2) integrating variability from an existing variability mechanism* into the presence conditions, and *3) building the variability model.*

4.3.1 Convert VCS Variability. To integrate the variability of the version-control system into the VariantInc platform, we collect all variable elements (i.e., files, lines, and byte arrays) in all variants from the VariantInc commit graph and compute their presence conditions. We begin by checking out the merge base commit from the version-control system to get all variable elements from the initial code base of the system. We add every file from the code base to the platform by storing its file path as a variable element. In addition, we determine for each file whether it is binary or textual in order to store its contents as variable elements as well (i.e., as a byte array or line-by-line, respectively). We assign a presence condition to each of these variable elements. During this initial stage, every presence condition consist of only the merge-base commit. We later update the presence condition when processing the next commits.

Next, we traverse the commit graph forwards in a breadth-first manner to iteratively build up the platform. For every commit in the graph, we add new variable elements and update existing presence conditions (e.g., when an element is deleted in a commit). Processing each commit results in a partial, yet already configurable, state of the VariantInc platform. For every commit, we compute the difference compared to the current state. The current state can be derived by configuring the platform and by selecting the predecessors of the current commit. Given a configuration, the current state consist of all *active* variable elements for this configuration. An element is active, if its presence condition evaluates to true for the given configuration, otherwise it is *inactive*.

The computed difference of the two states is file-based, which leads to three possible cases for every file. Each file was either *added*, *deleted*, or *modified*:

Adding a File. Analogous to the initial stage, we store the file path of the new file and either its binary or textual content. Every new presence condition consist of the current commit.

Deleting a File. When a file has been deleted, we do not remove it from the platform, but update its presences condition. In detail, we replace the old presence condition by a conjunction of the old presence condition and *not* the current commit. This ensures that if the current commit is selected in a configuration, the new presence condition will evaluate to false. Further, we update the presence conditions of all active variable elements of the file in the same way (i.e., deleting its lines or its byte array).

Modifying a File. For modifying a file, we update the presence conditions of its active variable elements. Modifying a file can be split into deleting and adding variable elements to a file. For binary files, we change the presence condition of its byte array by replacing it with a conjunction of the old presence condition and *not* the current commit (analogous to deleting a file). Further, we add the new byte array and create a new presence condition consisting of the current commit (analogous to adding a file). For textual files, we first iterate through all active lines and update the presence conditions of deleted lines accordingly. Next, we insert all added lines and update the presence conditions.

Via these means, we can represent the presence conditions for any variation in space and time that is caused by evolution in a version-control system.

4.3.2 *Integrate Existing Variability.*

For integrating the variability of an existing variability mechanism, we again analyze every variation from the VariantInc commit graph. We traverse the commit graph forwards with a breadth-first traversal and checkout every variation by configuring the platform. For every variation, we analyze the preprocessor annotations of every textual file. This allows us to determine presence conditions for every line regarding the preprocessor features. We then combine this new presence condition with the presence condition for the corresponding line. For every variation, we build a propositional implication, where the commit implies the preprocessor presence condition. If we did not yet add any implications to the old presence condition, we just add it by conjunction. Otherwise, we still add it, but update the other implications first by adding *and not the current commit* to the left side of every previous implication. This procedure guarantees that every line has the correct preprocessor presence condition for every configured variation.

4.3.3 *Build Variability Model.*

The complete variability model of the VariantInc platform consists of a conjunction of several models. First, we construct an initial variability model from the VariantInc commit graph. For every commit, we add a feature and a constraint that this commit implies its predecessors. Second, for every commit in the commit graph, we extract the corresponding variability model of the system. With these models, we then build and add implications to the VariantInc variability model, similar to integrating preprocessor presence conditions. Every model is implied by its commit and *not* any more recent commit that changes the model.

5 IMPLEMENTATION

We implemented VariantInc as an open-source prototype in Java. In this section, we describe the technological details of our prototype and its implementation that are independent of VariantInc itself.

Dependencies. Our prototype relies on external open-source libraries for analyzing forks and integrating them into a single platform. In detail, we use:

- JGit[2] for analyzing and traversing Git repositories;
- Eclipse EGit GitHub Connector[3] for communicating with the GitHub API;
- FeatureIDE[4] [32, 67], an open-source framework for creating propositional formulas for presence conditions; and
- FeatureCoPP[5] [53], which comprises a lightweight static analysis for extracting presence conditions of the CPP.

Our prototype is available via GitHub.[1]

Preparing Repositories. The first step for using VariantInc is to prepare the repository of the system for which forks shall be integrated (cf. Figure 1). To his end, we clone a system's Git repository, all publicly available branches and forks, and collect the required information. Instead of cloning each fork individually, technologically, we add each fork's remote to the repository and fetch its main branch. This allows us to access forks as if they were branches.

Handling Binary Files. A repository may include binary files, which are difficult to handle. In particular, due to Git replacing binary files completely after changes, we cannot analyze their individual variations, which would actually drastically increase the required memory space. To avoid such problems, we currently use a white list of file-extensions and a Git-like binary-file identification technique. We white list files with the following extensions, assuming that they represent text that VariantInc can analyze for variations in detail (recall that we focus on C systems): `c`, `h`, `cxx`, `hxx`, `cpp`, `hpp`, `txt`, `xml`, and `html`.

If a file does not match this white list, we try to automatically detect whether a file is a text or a binary file. For this purpose, we use JGit's internal heuristic that works similar to Git itself. We plan to extend our prototype so that a user can provide a custom white or black list to ensure a suitable identification of binary files.

[2]https://www.eclipse.org/jgit/
[3]https://github.com/eclipse/egit-github
[4]https://featureide.github.io/
[5]https://github.com/ldwxlnx/FeatureCoPP

Extracting Variability Models. The last step of the VariantInc construction process requires to extract the variability model of a system for every variation. Since extracting a variability model from a system is heavily dependent on the system's configuration mechanism, we were not able to implement a general automated extraction process. Thus, currently, we rely on the developers to provide an automated method for extracting their variability model.

Other Technical Limitations. We implemented our prototype of VariantInc focusing on Git and especially GitHub as version-control system, since it enables us to automatically collect all required information through libraries. So, while VariantInc itself can be applied to other version-control systems, our current prototype is not designed for this purpose. Besides this limitation of our prototype itself, some other problems may be caused by specifics of other version-control systems and could require individual solutions.

As we experienced during our validation (cf. Section 6), our prototype has a high memory consumption, particularly if VariantInc must create many and complex presence conditions. Consequently, our current implementation may run out of memory for extremely complex and large systems that exhibit too many forks (e.g., the Linux Kernel). However, this is not a limitation of VariantInc, but our current prototype, which has some potential to be optimized in this regard.

6 VALIDATION

We validated VariantInc by applying it on 160 open-source C systems. In this section, we describe our validation setup, the results we obtained, and discuss the outcomes.

6.1 Setup

Subject Systems. While implementing our prototype of VariantInc, we experimented with five subject systems (i.e., clamav, MPSolve, openvpn, subversion, and tcl) that have previously been used by Zhou et al. [74] for evaluating their technique for identifying features in forks. Using these five systems, we tested and optimized VariantInc. However, to show its feasibility and validate its behavior, we decided to use a variety of systems with different properties, particularly varying numbers of forks that could be integrated. For this purpose, we added GitHub projects to our analysis that use mainly C/C++ as programming language and had the most stars overall early in 2020. We remark that we had to exclude some systems with particularly many forks (e.g., Linux, Marlin) because we could not manage the number of forks on the hard drive we used. Precisely, we excluded systems with over 4,000 forks. Furthermore, we added the top 100 C/C++ system with most stars that had no more than 1,000 forks. In the end, we included 160 systems , for which we show examples and a statistical summary in Table 1 (the full overview is available in our repository[1]).

We can see that the systems we used for our validation span a wide range of properties, including smaller systems with few forks (e.g., axtls: 6 variants, 55 KLOC) up to large-scale systems (e.g., hhvm: 2,839 variants, 118 MLOC). The revision histories we analyzed range from 130 (i.e., HarmonyOS) up to 150,562 (i.e., tcl) commits. Moreover, we included systems from various domains, such as networking (e.g., openvpn), text editors (e.g., notepad-plus-plus), virtual machines (e.g., busybox), and version-control systems (e.g., subversion). So, our subject systems have diverse properties and are suitable to validate the feasibility of VariantInc for automatically integrating forks of different sizes and complexities into a configurable platform.

Validation System. To conduct our validation, we used a laptop with the following specifications: *CPU*: Intel Core i5-8350U, *RAM*: 16 GB, *OS*: Manjaro (Arch Linux), *Java*: OpenJDK 1.8.0_242, *JVM Memory*: Xmx: 14 GB, Xms: 2 GB.

Measurements. To validate VariantInc, we first checked that we could restore each of the previous forks by configuring the platform **(UC$_4$)** created fully automatically by VariantInc. More precisely, we configured the variations of each included fork and analyzed whether the resulting code base would be the same (i.e., we employed a line-by-line diff analysis). We did not measure the performance of VariantInc, since it is not our main concern and heavily depends on the system setup. For further validation and to provide an understanding how VariantInc can facilitate developers' tasks considering the use cases we motivated in Section 2.3, we measured several properties of each system—summarized in Table 1:

(1) We first show the **Variants** VariantInc analyzed, comprising the number of *Forks* (only the main branches of those) and *Branches* of the original repository. Moreover, we show the number of unique *Variations* that VariantInc identified, and the *Ratio* of these variations compared to all analyzed forks and branches. This ratio provides an intuition of how many actual variations exist for the analyzed systems in relation to its variants. Using this ratio, we can show to what extent our technique can facilitate the analysis of changes that may be integrated into a platform **(UC$_2$)**.
(2) We display an overview of the **Commits**, including the number of *Total* commits that existed for each system, the number of *NoOrphan* commits, and the number of *Pruned* commits that VariantInc identified to be relevant for the integration. These measurements indicate to what extent VariantInc simplifies and helps analyze revisions **(UC$_1$)**.
(3) We provide an overview of the **KLOC** of the systems and their forks. We show again the *Total* size of each system before the integration. Moreover, we show how many KLOC are *Active* on average in each possible configuration after the integration. The last value is a *Ratio* that indicates how similar the code of the systems and their forks actually is. So, a higher value indicates that the KLOC that are active in each configured variant comprise more variations identified in the total KLOC.
(4) We compare the **File Size** of each *Git* repository with all forks to the size of the platform created by *VariantInc.* The varying sizes provide an intuition about the integration VariantInc performs, and is mainly increasing if complex presence conditions must be added to the code base.

Using those measurements, we aimed to validate that VariantInc and its use cases are feasible for any system.

6.2 Results & Discussion

Now, we describe and discuss the measurements for all 160 systems we show in Table 1 according to the use cases we defined. Moreover,

Table 1: Examples and statistics of the 160 open-source systems on which we validated VariantInc.

System Name	Variants				Commits			KLOC			FileSize (MB)	
	Forks	*Branches*	*Variations*	*Ratio*	*Total*	*NoOrphans*	*Pruned*	*Total*	*∅ Active*	*Ratio*	*Git*	*VariantInc*
axtls	2	4	3	2.000	144	7	4	55.026	54.885	0.002	2.595	3.844
busybox	314	34	65	0.108	17,258	12,652	125	927.763	302.846	0.673	48.890	34.126
HarmonyOS	2,573	1	23	0.000	130	112	33	1,043.501	1,042.999	0.000	25.409	564.939
hhvm	2,733	106	266	0.038	45,412	44,614	522	18,155.693	3,036.375	0.832	494.865	1,775.119
notepad-plus-plus	2,707	1	223	0.000	5,423	3,376	384	1,216.568	441.953	0.636	260.512	1,104.383
openvpn	1,713	7	98	0.004	4,137	3,315	179	546.094	133.601	0.755	24.214	5.740
subversion	122	890	637	7.295	141,587	77,368	1,265	6,086.572	775.392	0.872	249.914	3,818.883
tcl	85	932	387	10.964	150,562	121,825	1,123	13,096.924	875.941	0.933	236.304	848.532
v8	2,859	3,592	1,933	1.256	106,143	86,392	3,986	11,194.959	2,706.649	0.758	963.755	575.645
Min	2	1	1	0.0002	27	1	1	2.228	1.727	0.0	0.144	0.027
Median	698.0	6.0	55.5	0.009	3,038.0	2,274.5	98.0	449.756	122.560	0.711	48.021	80.461
Mean	1,088.3	55.3	95.3	0.168	8,243.6	6,100.6	178.9	1,420.907	350.016	0.666	206.774	541.870
Max	3,944	3,592	1,933	10.964	150,562	121,825	3,986	2,1562.320	7,822.253	0.988	6,563.407	11,522.601

we discuss two hypotheses that relate to previous works, and which we could derive from the results we obtained using VariantInc. We remark that these hypotheses require additional studies and are not the focus of this paper, they only highlight the analysis use cases of using VariantInc.

Prune and Analyze Revision Histories of Forks (UC$_1$). As we can see in Table 1, the number of commits varies drastically between different systems and their forks. For example, tcl has 1,017 forks with 150,562 commits in total, whereas notepad-plus-plus has 2,707 forks with only 5,423 commits. A previous analysis of Marlin [65] shows that many forks of that system are inactive, used only for changing configuration files, and are rarely synchronized. So, for developers, but also for researchers, it is challenging to analyze such large numbers of forked systems and understand whether they comprise relevant variations. While GitHub intends to manage this complexity with pull-requests that can be submitted by developers of forks, this solution has limitations for organizations or projects that have to integrate a large number of forks, do not have such a mechanism, or develop a fork themselves.

VariantInc provides new capabilities for analyzing the revision histories, allowing developers and researchers to gain additional insights. For instance, considering the study of Stănciulescu et al. [65] and seeing our measurements, we could argue that forks may not be a good indicator for variations (and thus new features). More precisely, we can see in Table 1 that systems with a large number of variations also exhibit a large number of branches (e.g., subversion). In Figure 4, we compare the ratio of forks to branches in a system with the ratio of variations these exhibit for the system. So, dots on the left side of Figure 4 indicate systems with a higher number of branches compared to forks. For these systems, the number of variations is higher compared to such systems that exhibit more forks compared to branches.

We used Kendall's τ with the R statistics suite [59] to test whether this observation is meaningful. For this purpose, we compared the number of i) forks and ii) branches with the ratio of variations introduced by both together (i.e., *Variations*/(*Forks* + *Branches*)). Our null hypotheses were that neither forks nor branches would correlate with variations. Both tests yielded significant results (p-values < 0.001), indicating a negative tendency for forks (τ = -0.221) and a positive tendency for branches (τ = 0.256). While we can reject the null hypothesis, the effect sizes are small, wherefore we should not over-interpret the meaning. Still, it is interesting that the tendencies are opposing, indicating that branches may be were actual variations, such as new features and bug fixes, are implemented. This could indicate systems developed by an organization or a larger open-source community, or different development processes, for instance, branch-based (driven by the original developers) versus fork-based (driven by an open community) development. However, we require, and VariantInc provides the basis for, further analyses to investigate this hypothesis in more detail.

Hypothesis 1: branches versus forks

Our results indicate that there may be a difference in branch-based and fork-based development, since particularly branches seem to cause variations in a system.

Identify Variations in Forks (UC$_2$). We can see in Table 1 that VariantInc identified far fewer actual *Variations* than the number of forks and branches may have indicated, resulting in a relative small

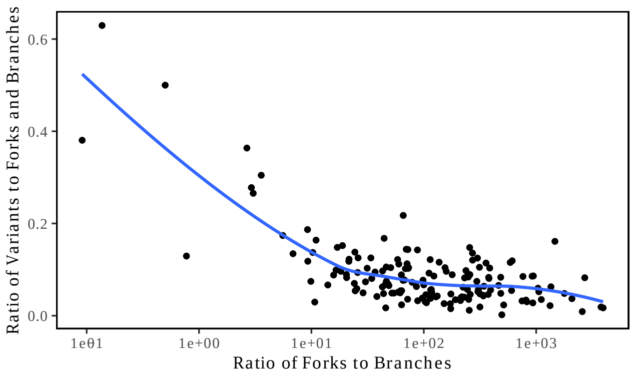

Figure 4: Comparing the ratio of forks to branches of a system with the ratio of variations VariantInc identified within these (logarithmic scale).

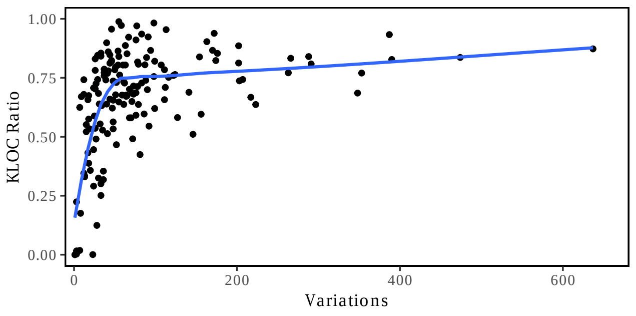

Figure 5: Comparing the number of identified variations with the ratio of variable KLOC.

Ratio. This again highlights that many forks and branches may not be actually developed, but only used as private copies. Moreover, considering the KLOC that are *Active* on average within a variant, we can see large differences. For instance, for axtls the variants mostly comprise the complete code base, resulting in a low ratio of variability. In contrast, a variant of tcl comprises on average 875 KLOC of the initially over 13 MLOC, resulting in a high variability ratio of 0.933.

Using VariantInc for such an analysis can help developers and researchers understand the complexity of variations that appear in forks. For instance, Liebig et al. [48] analyzed CPP usage in 40 systems, finding that the ratio of variability implemented with the CPP correlates to a system's size. Considering variations from forks and branches, it would not be surprising that a larger number of forks also results in more variations (even though our previous analysis puts this into perspective). However, aligning to the study of Liebig et al., it may be more interesting to understand how the system size relates to the possible variability after integrating variations.

In Figure 5, we compare the number of variations VariantInc identified in a system with the ratio of variability comprised in the resulting platform. We can see that both values seem to correlate, but except for a few outliers at the beginning, the effect does not seem to be strong. To analyze this observation, we again used Kendall's τ to test this hypothesis, with the null hypothesis being again that there is no correlation. The test reveals a significant ($p < 0.001$) positive tendency ($\tau = 0.358$). So, while more variations result, not surprisingly, in more variability, it is interesting that the value seems to be rather stable except for some outliers.

Hypothesis 2: ratio of variability

Our results indicate that the percentage of variability introduced by integrating forks remains relatively stable compared to a the systems' sizes.

Automatically Integrate Forks into a Platform (UC$_3$). For now, we successfully validated that our technique can integrate all existing forks and branches of a system into a new platform. An adaptation for a version-control system is possible, since our variant integration works as intended.

Configure Variations in Space and Time (UC$_4$). We configured the resulting VariantInc platform of each system to derive each of the previously existing forks. A diff analysis of each of the configured variants showed that they were all identical to their original forks (i.e., after integrating all variations, we can still derive the integrated system variants), confirming that VariantInc integrates all variations as intended. So, the resulting VariantInc platform could be used directly, but arguably developers should first assign proper feature names to the presence conditions, adapt the variability mechanism (e.g., integrate new features into their configurator tool), and refactor some code changes. Still, these are expected activities to improve the quality of a re-engineered software system [1, 15, 33, 35, 41, 46], and our technique considerably facilitates such processes.

Migrate towards Variation Control (UC$_5$). As motivated, we did not validate this use case, due to its technical restrictions. However, considering the previous results, we can see that our technique does provide all artifacts required to migrate systems towards variation control. We aim to address this migration in future work.

7 CONCLUSION

In this paper, we presented VariantInc, a technique for analyzing and integrating forks of a variant-rich system into a platform that also considers revision histories. Moreover, VariantInc extends existing analysis methods for forked systems and their revisions, allowing for novel analyses of software-engineering practices. We validated that VariantInc works properly and as intended by employing it on 160 open-source systems. Based on the results, we discussed VariantInc's feasibility for five different use cases that have not been well supported before. In summary, we show that:

- VariantInc allows for novel analyses of revision histories of variant-rich systems that can be used to reveal new insights regarding the evolution of such systems.
- VariantInc can help developers to identify and locate the unique variations that exist in different forks.
- VariantInc is feasible to integrate the forks of a wide range of highly complex systems form various domains.
- VariantInc results in a configurable platform that can be used directly to derive system variants, even though refinements by developers are needed to align the platform to established processes (e.g., assign features).

VariantInc is available as an open-source prototype[1] and may be extended and integrated into other tools. We hope that our technique can help practitioners in facilitating the management of variant-rich systems, and researchers in providing new analysis methods and research opportunities.

In future work, we plan to integrate VariantInc into tool frameworks for managing variant-rich systems, providing interfaces to different version-control and variation-control systems. We aim to conduct empirical evaluations and user studies of how VariantInc can support developers in the different use cases we described, and are particularly interested in advancing the support for variation-control. Moreover, we found that VariantInc can be used for novel analyses that could reveal interesting and important properties of variant-rich systems and their development. So, we aim to study such properties in more detail and based on larger sets of systems, also involving other types of empirical studies to confirm our findings qualitatively and quantitatively.

ACKNOWLEDGMENTS

The research reported in this paper has been partially supported by the German Research Foundation (DFG) projects EXPLANT (LE 3382/2-3, SA 465/49-3) [40] and VariantSync (TH 2387/1-1) [30].

REFERENCES

[1] Jonas Åkesson, Sebastian Nilsson, Jacob Krüger, and Thorsten Berger. 2019. Migrating the Android Apo-Games into an Annotation-Based Software Product Line. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 103–107. https://doi.org/10.1145/3336294.3342362

[2] Vander Alves, Pedro Matos Jr., Leonardo Cole, Paulo Borba, and Geber Ramalho. 2005. Extracting and Evolving Mobile Games Product Lines. In *International Software Product Line Conference (SPLC)*. Springer, 70–81. https://doi.org/10.1007/11554844_8

[3] Sofia Ananieva, Sandra Greiner, Timo Kehrer, Jacob Krüger, Thomas Kühn, Lukas Linsbauer, Sten Grüner, Anne Koziolek, Henrik Lönn, S. Ramesh, and Ralf Reussner. 2022. A Conceptual Model for Unifying Variability in Space and Time: Rationale, Validation, and Illustrative Applications. *Empirical Software Engineering* 27, 101 (2022), 1–53. https://doi.org/10.1007/s10664-021-10097-z

[4] Sofia Ananieva, Sandra Greiner, Thomas Kühn, Jacob Krüger, Lukas Linsbauer, Sten Grüner, Timo Kehrer, Heiko Klare, Anne Koziolek, Henrik Lönn, Sebastian Krieter, Christoph Seidl, S. Ramesh, Ralf Reussner, and Bernhard Westfechtel. 2020. A Conceptual Model for Unifying Variability in Space and Time. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 15:1–12. https://doi.org/10.1145/3382025.3414955

[5] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer. https://doi.org/10.1007/978-3-642-37521-7

[6] Wesley K. G. Assunção, Roberto E. Lopez-Herrejon, Lukas Linsbauer, Silvia R. Vergilio, and Alexander Egyed. 2017. Reengineering Legacy Applications into Software Product Lines: A Systematic Mapping. *Empirical Software Engineering* 22, 6 (2017), 2972–3016. https://doi.org/10.1007/s10664-017-9499-z

[7] Veronika Bauer and Antonio Vetro'. 2016. Comparing Reuse Practices in Two Large Software-Producing Companies. *Journal of Systems and Software* 117 (2016), 545–582. https://doi.org/10.1016/j.jss.2016.03.067

[8] Thorsten Berger, Marsha Chechik, Timo Kehrer, and Manuel Wimmer (Eds.). 2019. *Software Evolution in Time and Space: Unifying Version and Variability Management*. Schloss Dagstuhl.

[9] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 1–8. https://doi.org/10.1145/2430502.2430513

[10] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. 1993. The Concept Assignment Problem in Program Understanding. In *Working Conference on Reverse Engineering (WCRE)*. IEEE, 482–498. https://doi.org/10.1109/wcre.1993.287781

[11] Paul C. Clements and Charles W. Krueger. 2002. Point/Counterpoint: Being Proactive Pays Off / Eliminating the Adoption Barrier. *IEEE Software* 19, 4 (2002), 28–30. https://doi.org/10.1109/ms.2002.1020283

[12] Reidar Conradi and Bernhard Westfechtel. 1998. Version Models for Software Configuration Management. *ACM Computing Surveys* 30, 2 (1998), 232–282. https://doi.org/10.1145/280277.280280

[13] Marcus V. Couto, Marco T. Valente, and Eduardo Figueiredo. 2011. Extracting Software Product Lines: A Case Study Using Conditional Compilation. In *European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 191–200. https://doi.org/10.1109/csmr.2011.25

[14] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. 2012. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 173–182. https://doi.org/10.1145/2110147.2110167

[15] Jamel Debbiche, Oskar Lignell, Jacob Krüger, and Thorsten Berger. 2019. Migrating Java-Based Apo-Games into a Composition-Based Software Product Line. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 98–102. https://doi.org/10.1145/3336294.3342361

[16] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2011. Feature Location in Source Code: A Taxonomy and Survey. *Journal of Software: Evolution and Process* 25, 1 (2011), 53–95. https://doi.org/10.1002/smr.567

[17] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 25–34. https://doi.org/10.1109/csmr.2013.13

[18] Slawomir Duszynski, Jens Knodel, and Martin Becker. 2011. Analyzing the Source Code of Multiple Software Variants for Reuse Potential. In *Working Conference on Reverse Engineering (WCRE)*. IEEE, 303–307. https://doi.org/10.1109/wcre.2011.44

[19] Wolfram Fenske, Jens Meinicke, Sandro Schulze, Steffen Schulze, and Gunter Saake. 2017. Variant-Preserving Refactorings for Migrating Cloned Products to a Product Line. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 316–326. https://doi.org/10.1109/saner.2017.7884632

[20] Wolfram Fenske, Thomas Thüm, and Gunter Saake. 2013. A Taxonomy of Software Product Line Reengineering. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 4:1–8. https://doi.org/10.1145/2556624.2556643

[21] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 391–400. https://doi.org/10.1109/icsme.2014.61

[22] Thomas Fogdal, Helene Scherrebeck, Juha Kuusela, Martin Becker, and Bo Zhang. 2016. Ten Years of Product Line Engineering at Danfoss: Lessons Learned and Way Ahead. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 252–261. https://doi.org/10.1145/2934466.2934491

[23] Cristina Gacek and Michalis Anastasopoules. 2001. Implementing Product Line Variabilities. In *Symposium on Software Reusability (SSR)*. ACM, 109–117. https://doi.org/10.1145/375212.375269

[24] Georgios Gousios, Martin Pinzger, and Arie van Deursen. 2014. An Exploratory Study of the Pull-Based Software Development Model. In *International Conference on Software Engineering (ICSE)*. ACM, 345–355. https://doi.org/10.1145/2568225.2568260

[25] William A. Hetrick, Charles W. Krueger, and Joseph G. Moore. 2006. Incremental Return on Incremental Investment: Engenio's Transition to Software Product Line Practice. In *International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. ACM, 798–804. https://doi.org/10.1145/1176617.1176726

[26] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßenich, Martin Becker, and Sven Apel. 2015. Preprocessor-Based Variability in Open-Source and Industrial Software Systems: An Empirical Study. *Empirical Software Engineering* 21, 2 (2015), 449–482. https://doi.org/10.1007/s10664-015-9360-1

[27] Hans P. Jepsen, Jan G. Dall, and Danilo Beuche. 2007. Minimally Invasive Migration to Software Product Lines. In *International Software Product Line Conference (SPLC)*. IEEE, 203–211. https://doi.org/10.1109/spline.2007.30

[28] Christian Kästner, Sven Apel, and Don Batory. 2007. A Case Study Implementing Features Using AspectJ. In *International Software Product Line Conference (SPLC)*. IEEE, 223–232. https://doi.org/10.1109/spline.2007.12

[29] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. ACM, 805–824. https://doi.org/10.1145/2048066.2048128

[30] Timo Kehrer, Thomas Thüm, Alexander Schultheiß, and Paul Maximilian Bittner. 2021. Bridging the Gap Between Clone-and-Own and Software Product Lines. In *International Conference on Software Engineering (ICSE)*. IEEE, 21–25. https://doi.org/10.1109/ICSE-NIER52604.2021.00013

[31] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina V. Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 220–242. https://doi.org/10.1007/bfb0053381

[32] Sebastian Krieter, Marcus Pinnecke, Jacob Krüger, Joshua Sprey, Christopher Sontag, Thomas Thüm, Thomas Leich, and Gunter Saake. 2017. FeatureIDE: Empowering Third-Party Developers. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 42–45. https://doi.org/10.1145/3109729.3109751

[33] Jacob Krüger. 2021. *Understanding the Re-Engineering of Variant-Rich Systems: An Empirical Work on Economics, Knowledge, Traceability, and Practices.* Ph.D. Dissertation. Otto-von-Guericke University Magdeburg. https://doi.org/10.25673/39349

[34] Jacob Krüger, Mustafa Al-Hajjaji, Sandro Schulze, Gunter Saake, and Thomas Leich. 2018. Towards Automated Test Refactoring for Software Product Lines. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 143–148. https://doi.org/10.1145/3233027.3233040

[35] Jacob Krüger and Thorsten Berger. 2020. Activities and Costs of Re-Engineering Cloned Variants Into an Integrated Platform. In *International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 21:1–10. https://doi.org/10.1145/3377024.3377044

[36] Jacob Krüger and Thorsten Berger. 2020. An Empirical Analysis of the Costs of Clone- and Platform-Oriented Software Reuse. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 432–444. https://doi.org/10.1145/3368089.3409684

[37] Jacob Krüger, Thorsten Berger, and Thomas Leich. 2019. Features and How to Find Them - A Survey of Manual Feature Location. In *Software Engineering for Variability Intensive Systems*. CRC Press, 153–172. https://doi.org/10.1201/9780429022067-9

[38] Jacob Krüger, Wolfram Fenske, Jens Meinicke, Thomas Leich, and Gunter Saake. 2016. Extracting Software Product Lines: A Cost Estimation Perspective. In

International Systems and Software Product Line Conference (SPLC). ACM, 354–361. https://doi.org/10.1145/2934466.2962731

[39] Jacob Krüger, Wolfram Fenske, Thomas Thüm, Dirk Aporius, Gunter Saake, and Thomas Leich. 2018. Apo-Games - A Case Study for Reverse Engineering Variability from Cloned Java Variants. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 251–256. https://doi.org/10.1145/3233027.3236403

[40] Jacob Krüger, Sebastian Krieter, Gunter Saake, and Thomas Leich. 2020. EXtracting Product Lines from vAriaNTs (EXPLANT). In *International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 13:1–2. https://doi.org/10.1145/3377024.3377046

[41] Jacob Krüger, Wardah Mahmood, and Thorsten Berger. 2020. Promote-pl: A Round-Trip Engineering Process Model for Adopting and Evolving Product Lines. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 2:1–12. https://doi.org/10.1145/3382025.3414970

[42] Jacob Krüger, Alex Mikulinski, Sandro Schulze, Thomas Leich, and Gunter Saake. 2023. DSDGen: Extracting Documentation to Comprehend Fork Merges. In *International Systems and Software Product Line Conference (SPLC)*. ACM.

[43] Jacob Krüger, Mukelabai Mukelabai, Wanzi Gu, Hui Shen, Regina Hebig, and Thorsten Berger. 2019. Where is My Feature and What is it About? A Case Study on Recovering Feature Facets. *Journal of Systems and Software* 152 (2019), 239–253. https://doi.org/10.1016/j.jss.2019.01.057

[44] Jacob Krüger, Louis Nell, Wolfram Fenske, Gunter Saake, and Thomas Leich. 2017. Finding Lost Features in Cloned Systems. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 65–72. https://doi.org/10.1145/3109729.3109736

[45] Elias Kuiter, Sebastian Krieter, Jacob Krüger, Kai Ludwig, Thomas Leich, and Gunter Saake. 2018. PCLocator: A Tool Suite to Automatically Identify Configurations for Code Locations. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 284–288. https://doi.org/10.1145/3233027.3236399

[46] Elias Kuiter, Jacob Krüger, Sebastian Krieter, Thomas Leich, and Gunter Saake. 2018. Getting Rid of Clone-And-Own: Moving to a Software Product Line for Temperature Monitoring. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 189–189. https://doi.org/10.1145/3233027.3233050

[47] Miguel A. Laguna and Yania Crespo. 2013. A Systematic Mapping Study on Software Product Line Evolution: From Legacy System Reengineering to Product Line Refactoring. *Science of Computer Programming* 78, 8 (2013), 1010–1034. https://doi.org/10.1016/j.scico.2012.05.003

[48] 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 *International Conference on Software Engineering (ICSE)*. ACM, 105–114. https://doi.org/10.1145/1806799.1806819

[49] Max Lillack, Ştefan Stănciulescu, Wilhelm Hedman, Thorsten Berger, and Andrzej Wąsowski. 2019. Intention-Based Integration of Software Variants. In *International Conference on Software Engineering (ICSE)*. IEEE, 831–842. https://doi.org/10.1109/icse.2019.00090

[50] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. 2017. A Classification of Variation Control Systems. In *International Conference on Generative Programming: Concepts and Experiences (GPCE)*. ACM, 49–62. https://doi.org/10.1145/3136040.3136054

[51] Jia Liu, Don Batory, and Christian Lengauer. 2006. Feature Oriented Refactoring of Legacy Applications. In *International Conference on Software Engineering (ICSE)*. ACM, 112–121. https://doi.org/10.1145/1134285.1134303

[52] John Long. 2001. Software Reuse Antipatterns. *ACM SIGSOFT Software Engineering Notes* 26, 4 (2001), 68–76. https://doi.org/10.1145/505482.505492

[53] Kai Ludwig, Jacob Krüger, and Thomas Leich. 2020. FeatureCoPP: Unfolding Preprocessor Variability. In *International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 24:1–9. https://doi.org/10.1145/3377024.3377039

[54] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2015. Bottom-Up Adoption of Software Product Lines - A Generic and Extensible Approach. In *International Software Product Line Conference (SPLC)*. ACM, 101–110. https://doi.org/10.1145/2791060.2791086

[55] Sarah Nadi, Thorsten Berger, Christian Kastner, and Krzysztof 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. https://doi.org/10.1109/tse.2015.2415793

[56] Damir Nešić, Jacob Krüger, Ştefan Stănciulescu, and Thorsten Berger. 2019. Principles of Feature Modeling. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 62–73. https://doi.org/10.1145/3338906.3338974

[57] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering*. Springer. https://doi.org/10.1007/3-540-28901-1

[58] Christian Prehofer. 1997. Feature-Oriented Programming: A Fresh Look at Objects. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 419–443. https://doi.org/10.1007/bfb0053389

[59] R Core Team. 2020. *R: A Language and Environment for Statistical Computing*. https://www.R-project.org

[60] Julia Rubin and Marsha Chechik. 2013. A Survey of Feature Location Techniques. In *Domain Engineering*. Springer, 29–58. https://doi.org/10.1007/978-3-642-36654-3_2

[61] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. 2015. Cloned Product Variants: From Ad-Hoc to Managed Software Product Lines. *International Journal on Software Tools for Technology Transfer* (2015), 627–646. https://doi.org/10.1007/s10009-014-0347-9

[62] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. 2012. Software Diversity: State of the Art and Perspectives. *International Journal on Software Tools for Technology Transfer* 14, 5 (2012), 477–495. https://doi.org/10.1007/s10009-012-0253-y

[63] Klaus Schmid and Martin Verlage. 2002. The Economic Impact of Product Line Adoption and Evolution. *IEEE Software* 19, 4 (2002), 50–57. https://doi.org/10.1109/ms.2002.1020287

[64] Sandro Schulze, Jacob Krüger, and Johannes Wünsche. 2022. Towards Developer Support for Merging Forked Test Cases. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 131–141. https://doi.org/10.1145/3546932.3547002

[65] Ştefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. 2015. Forked and Integrated Variants in an Open-Source Firmware Project. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 151–160. https://doi.org/10.1109/icsm.2015.7332461

[66] Daniel Strüber, Mukelabai Mukelabai, Jacob Krüger, Stefan Fischer, Lukas Linsbauer, Jabier Martinez, and Thorsten Berger. 2019. Facing the Truth: Benchmarking the Techniques for the Evolution of Variant-Rich Systems. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 177–188. https://doi.org/10.1145/3336294.3336302

[67] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming* 79 (2014), 70–85. https://doi.org/10.1016/j.scico.2012.06.002

[68] Marco T. Valente, Virgilio Borges, and Leonardo Passos. 2012. A Semi-Automatic Approach for Extracting Software Product Lines. *IEEE Transactions on Software Engineering* 38, 4 (2012), 737–754. https://doi.org/10.1109/tse.2011.57

[69] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. 2007. *Software Product Lines in Action*. Springer. https://doi.org/10.1007/978-3-540-71437-8

[70] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. 2013. How Developers Perform Feature Location Tasks: A Human-Centric and Process-Oriented Exploratory Study. *Journal of Software: Evolution and Process* 25, 11 (2013), 1193–1224. https://doi.org/10.1002/smr.1593

[71] Norman Wilde, Michelle Buckellew, Henry Page, Vaclav Rajlich, and LaTreva Pounds. 2003. A Comparison of Methods for Locating Features in Legacy Software. *Journal of Systems and Software* 65, 2 (2003), 105–114. https://doi.org/10.1016/s0164-1212(02)00052-3

[72] Yinxing Xue. 2011. Reengineering Legacy Software Products into Software Product Line Based on Automatic Variability Analysis. In *International Conference on Software Engineering (ICSE)*. ACM, 1114–1117. https://doi.org/10.1145/1985793.1986009

[73] Kentaro Yoshimura, Dharmalingam Ganesan, and Dirk Muthig. 2006. Assessing Merge Potential of Existing Engine Control Systems into a Product Line. In *International Workshop on Software Engineering for Automotive Systems (SEAS)*. ACM, 61–67. https://doi.org/10.1145/1138474.1138485

[74] Shurui Zhou, Ştefan Stănciulescu, Olaf Leßenich, Yingfei Xiong, Andrzej Wąsowski, and Christian Kästner. 2018. Identifying Features in Forks. In *International Conference on Software Engineering (ICSE)*. ACM, 106–116. https://doi.org/10.1145/3180155.3180205