Refactoring Product Lines by Replaying Version Histories

Michael Ratzenböck
Paul Grünbacher
Wesley Klewerton Guez Assunção
Alexander Egyed
Institute of Software Systems Engineering
Johannes Kepler University Linz
4040 Linz, Austria
paul.gruenbacher@jku.at

Lukas Linsbauer
Institute of Software Engineering and Automotive Informatics
Technische Universität Braunschweig
38106 Braunschweig, Germany
l.linsbauer@tu-braunschweig.de

ABSTRACT

When evolving software product lines, new features are added over time and existing features are revised. Engineers also decide to merge different features or split features in other cases. Such refactoring tasks are difficult when using manually maintained feature-to-code mappings. Intensional version control systems such as ECCO overcome this issue with automatically computed feature-to-code mappings. Furthermore, they allow creating variants that have not been explicitly committed before. However, such systems are still rarely used compared to extensional version control systems like Git, which keep track of the evolution history by assigning revisions to states of a system. This paper presents an approach combining both extensional and intensional version control systems, which relies on the extensional version control system Git to store versions. Developers selectively tag existing versions to describe the evolution at the level of features. Our approach then automatically replays the evolution history to create a repository of the intensional variation control system ECCO. The approach contributes to research on refactoring features of existing product lines and migrating existing systems to product lines. We provide an initial evaluation of the approach regarding correctness and performance based on an existing system.

CCS CONCEPTS

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

KEYWORDS

version control systems, refactoring, feature-level evolution

ACM Reference Format:
Michael Ratzenböck, Paul Grünbacher, Wesley Klewerton Guez Assunção, Alexander Egyed, and Lukas Linsbauer. 2022. Refactoring Product Lines by Replaying Version Histories. In *Proceedings of the 16th International Working Conference on Variability Modelling of Software-Intensive Systems (VAMOS '22), February 23–25, 2022, Florence, Italy*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3510466.3510484

1 INTRODUCTION

Product lines are subject to continuous evolution [13, 23]. Features are added, removed, and renamed over time, and they are split or merged to accommodate new or changing requirements [3]. These continuous changes result in many revisions of software artifacts [7]. Revisions are the result of evolution in time, e.g., when fixing a bug. They denote sequential versions, representing a snapshot of the evolution of a software feature. Variants on the other hand stem from evolution in space [2], e.g., when adding a new feature. They denote versions of software artifacts that need to exist concurrently. In annotation-based product lines engineers manually maintain feature-to-code mappings. Maintaining code fragments guarded by annotations encoding the mappings is hard [15, 21] and it is particularly challenging to carry out changes to features while at the same time keeping the mappings consistent [13, 22, 28]. For instance, merging features at a certain point is difficult when done manually, since features are mapped to diverse and complex artifacts.

Existing version control systems pursue two versioning strategies [7, 18], which can be used to manage evolving product lines: *Extensional versioning* assumes that all existing versions are explicitly enumerated. It then allows to retrieve the versions that have been created before. Git or Subversion are examples of such tools, which keep track of changes by assigning revisions to states of a system over time. However, evolution is rarely just a linear sequence of steps and such tools thus provide branching mechanisms for dealing with variants. For instance, short-term branches are used to develop new features in isolation. Once a new feature is finished, it is merged with the original artifact and the branch is no longer used. However, at this point the new feature becomes tangled with the rest of the artifacts and its location is not managed explicitly [22]. The purpose of long-term branches, on the other hand, is to create clones of existing artifacts, based on which variants of the system can then be created. Nonetheless, long-term branches quickly lead to maintenance problems as updates and fixes need to be propagated to all variants [25]. *Intensional versioning* aims at overcoming these limitations with mechanisms for managing fine-grained variants [18], thereby avoiding branches for features of variants. Furthermore, they allow creating versions that have not been explicitly enumerated and committed before. Such tools use concepts like features, configurations, and construction

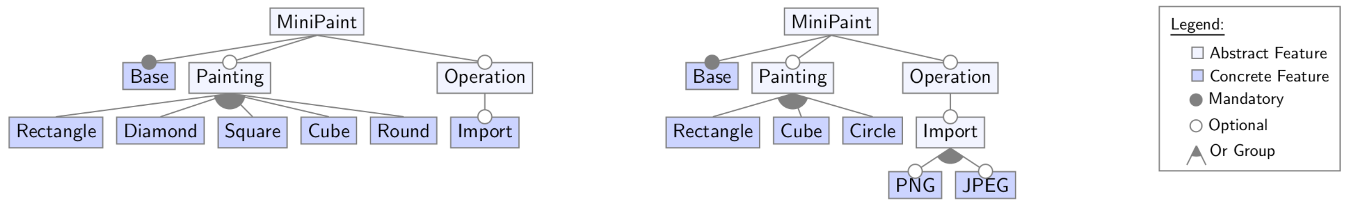

Figure 1: The original (left) and the refactored (right) feature model of MiniPaint.

rules to compose arbitrary versions [3, 8]. Most importantly, intensional version control systems automatically compute and update mappings between features and artifacts when engineers commit features to a repository. Examples of such tools are ECCO [16] and SuperMod [18].

The current state of the practice is to use extensional tools such as Git combined with annotation-based variability mechanisms such as preprocessors to manage both revisions and variants of software systems [22, 23, 26]. Despite its popularity, this approach lacks the ability of creating new variants based on arbitrary feature combinations [2]. Furthermore, it requires developers to manually edit feature annotations [22], which could be automated by variation control systems as pointed out. This paper thus combines extensional and intensional version control to benefit from both kinds of approaches for the purpose of refactoring product lines or migrating systems to product lines.

In particular, we claim the following contributions: *(i)* we present an approach based on lightweight feature tags to selectively annotate existing version histories. Our tool-supported approach then analyzes and replays the tagged version history to create a repository of a variation control system, which allows managing both revisions and variants. *(ii)* We provide an implementation, which uses the version control system Git to manage snapshots of the evolution and the variation control system ECCO [10, 16, 17] to manage both revisions and variants. *(iii)* We evaluate our approach regarding correctness and performance by applying it to the MagicMirror system and its evolution history. *(iv)* Finally, we discuss experiences and lessons learned.

2 MOTIVATING EXAMPLE

As a running example, let us consider the original and the refactored feature model of the simple MiniPaint application (cf. Figure 1) for painting graphical objects. When evolving MiniPaint an engineer may consider refactoring the model. For example, drawing a rectangle is similar to drawing a diamond or a square, thus, engineers might *merge* these features to a single feature Rectangle, thereby reducing complexity. Moreover, engineers might decide to *rename* the feature Round, which is responsible for drawing circles. Furthermore, the feature Import that handles the import of different file types may get too complex. Hence, engineers may propose to *split* it into multiple features to reduce feature complexity and to offer additional choices to customers.

The example highlights that evolving a software product line will require adding new features and revising existing ones over time, thereby providing more functionality or changing feature behavior, as recently shown in empirical studies on the evolution of highly configurable systems [22, 23]. Furthermore, there might be a point in time when different features provide very similar functionality and it makes sense to merge them. The initial meaning of features may even evolve over time, making it necessary to refactor the originally anticipated model [3].

Such refactoring tasks are challenging when using manually maintained preprocessor annotations to manage feature-to-code mappings. For instance, developers will have to go through the code base and change all affected annotations. This is both tedious and error-prone, as shown by Michelon et al. [22, 23], who mined features from existing Git repositories. Variation control systems [18] overcome this issue by automatically computing feature-to-code mappings. For instance, ECCO computes feature-to-code mappings by determining feature-level differences between different artifact versions in the code base [10, 17]. This allows to refactor product lines by replaying an evolution history (e.g., from a Git repository) and committing the different versions with the desired features to a variation control system.

3 THE RESERVE APPROACH

We present the ReSeRVe (**Re**factoring by **Se**lectively **R**eplaying **Ve**rsions) approach that integrates both extensional and intensional version control to utilize the benefits of both kinds of version control systems. Our approach comprises five elements (cf. Figure 2):

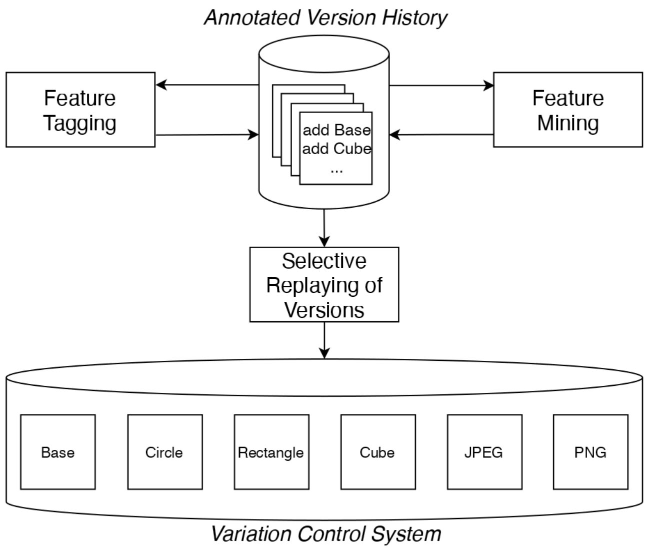

Figure 2: Overview of the ReSeRVe approach.

It relies on manual Feature Tagging to annotate existing version histories and/or automated Feature Mining, such as shown by [23]. ReSeRVe then automatically replays the Annotated Evolution History and commits the feature-level changes to a Variation Control System, thereby creating a repository which manages both revisions and variants at the level of features.

Annotated Version History. The input for our automated approach is an annotated version history. We do not make assumptions on how the evolution snapshots of existing systems are managed, as long as we can re-create (or check out) earlier system versions, which is the case for tools like Git or Subversion. The evolution history can be annotated manually by feature tagging, i.e., describing existing commits with feature-level changes. Alternatively, an existing repository can be analyzed and annotated automatically using a feature mining approach, as shown in related work [22, 23]. The mined features and their evolution history can then be used as additional input for manual tagging.

Feature Tagging. In this manual process, an engineer annotates versions, i.e., commits, of the evolution history. The annotations encode information about features and the refactoring operations applied on these features, i.e., they describe when features were first added, updated, or deleted. Feature operations also include merging multiple features into a single one, splitting a single feature into multiple features, or renaming a feature, as already discussed in the motivating example. Manual tagging is straightforward with existing tools such as Atlassian's SourceTree, gitk, or TortoiseSVN, which provide a graphical user interfaces besides a command line interface to tag the version history.

Feature Mining. Feature annotations based on preprocessor directives are a common variability mechanism used in product lines and highly-configurable software systems [26]. However, it is challenging to understand, maintain, and evolve code fragments guarded by `#ifdef` directives encoding the feature-to-code mappings [15, 21]. Automated approaches have been developed to mine and analyze features and their life cycle from existing code repositories [22, 23]. They can be used to support feature tagging by analyzing system evolution at the level of features, e.g., by showing when features were first introduced, or when they were revised.

Variation Control System. Several variation control systems with different capabilities have been proposed [18]. When working with a variation control system, a developer uses a feature-oriented checkout operation to create a variant and then implements new features or makes changes to existing features. The developer then uses a feature-oriented commit operation to submit changes to the repository by providing information which features have been added or changed. The system determines the changes in the implementation artifacts and updates the mappings of changed implementation artifacts to new features and feature revisions. If the committed code artifacts affect already existing features, a new feature revision is created automatically [13, 18].

Selective Replaying of Versions. This automated step analyzes the version history annotated with information about feature changes to generate a repository of a variation control system. This is achieved by selectively replaying the evolution history and using the annotations to incrementally generate a new repository by automatically creating and executing commits to a variation control system.

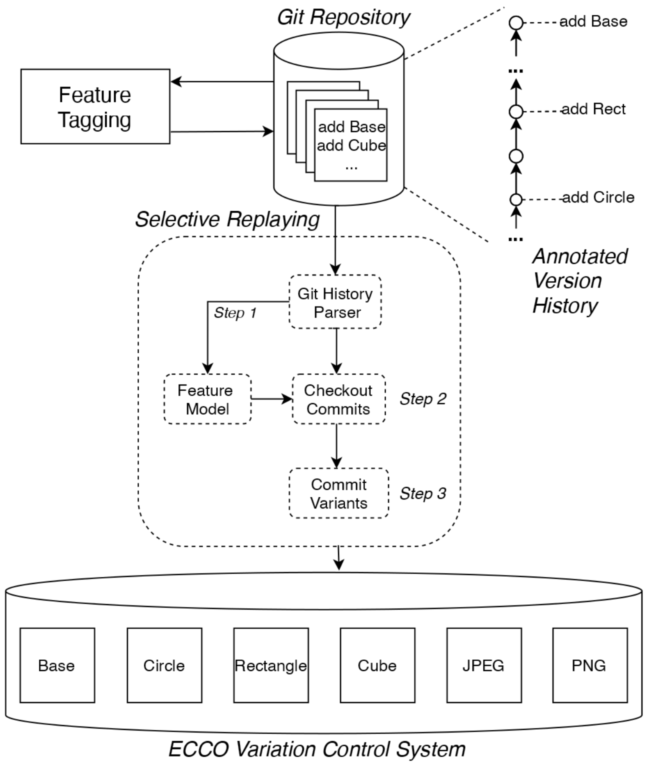

Figure 3: ReSeRVe tool architecture.

4 IMPLEMENTATION

The tool architecture of ReSeRVe is depicted in Figure 3. It realizes the general approach shown in Figure 2. To limit the scope of this paper, our current implementation and evaluation focuses on manual tagging, while we will complement the approach with automated feature mining in future work.

4.1 Capabilities

Our implementation of ReSeRVe relies on selectively annotating a Git history via feature-level tags and then generating a repository of the variation control system ECCO [16], based on the provided tags. Annotated tags have both a name and a message defining the operation for feature refactoring. An engineer describes changes at the level of features by adding or removing tags, either via the Git command line interface or by using a Git GUI client. The tags are stored in the Git repository. While Git does not provide support to refactor product lines at the level of features, Git tags provide a simple mechanism to annotate the evolution history by specifying feature operations. This also ensures traceability, as, for example, adding a new feature is linked to a commit and a tag. Further, by inspecting the Git history, an engineer can quickly spot when a feature was added, revised, renamed, or deleted.

Our tool then performs the following three steps:

Step 1 – Parsing a tagged Git history and generating a feature model. This step filters out tags containing unsupported refactoring operations and performs basic consistency checks on the found features. For example, it is important to prevent merging a feature that does not yet exist at a certain point in time.

Step 2 – Checking out relevant commits. This step uses the generated feature model to re-parse the Git history. The tool computes the feature versions and selects commits with relevant tags for checking them out from the repository. The tool also computes the feature configurations of each version based on the feature model.

Step 3 – Committing the checked-out versions. In this step, the tool incrementally commits all the checked-out versions using their respective feature configurations to the ECCO repository. This is done for every checked-out commit.

4.2 Tagging Support

Our tool supports six operations for defining feature-level changes, which can be added to tags when analyzing a version history:

Add. This operation indicates that a feature has been first introduced in a particular commit. This can be seen as the entry point for managing a new feature. If a feature already exists, it is considered a revision and its version number is increased. As shown by Michelon et al. [22] commits frequently concern existing features but often also affect new ones.

Update. This operation allows to explicitly revise an already existing feature, similar to adding an existing one. The operation only succeeds if the feature already exists.

Rename. This operation can be used if a feature name is no longer considered appropriate in the domain of the product line. The new feature name must be unique.

Merge. This operation allows combining multiple features, e.g., if they are highly dependent on each other or provide very similar functionality. This operation replaces all occurrences of the features and uses the new name of the feature from the commit of the merge onwards.

Split. This operation divides a single feature into multiple features sharing the same base. This is useful, for instance, if a feature gets too complex or provides too many capabilities. Essentially, the feature to be split is replaced with all its subfeatures. Obviously, this operation requires additional commits tagged with the subfeatures in the subsequent version history such that the variation control system can distinguish them as we will discuss later.

Delete. This operation removes a feature from being tracked, meaning that it will not appear in the variation control system and thus can no longer be used to create variants. When deleting a feature the affected commits in the version history are either associated with the base feature or with other features. Specifically, if no features are left in a tag, the base feature will be updated, otherwise the remaining features will be mapped to the code.

Figure 4 shows the Git history and the feature evolution graph for our running example. It illustrates how the change operations like *merge, rename* and *split* cause features to *evolve.* When a feature evolves, it is not present anymore in the final feature model. For instance, the feature *Rect* is added in commit #1. However, this feature is not present in the final commit #6, as it evolved into *Square* and *Diamond.* This also means that the *Rect* feature will not be available in the generated ECCO repository. However, the original tag message is not removed, since it describes the original introduction of the features and the user is not assumed to manually replace feature *Rect* with its subfeatures in the annotated version history.

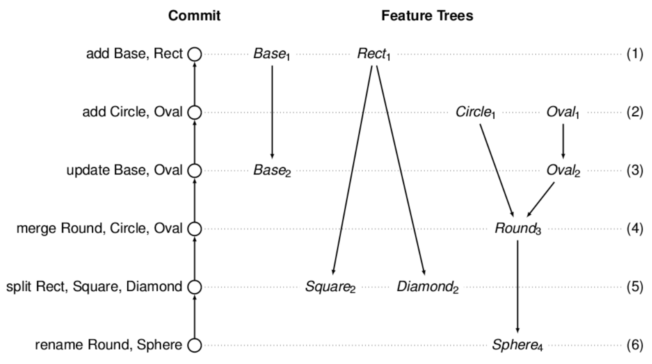

Figure 4: MiniPaint commits with feature-level refactorings. The example also shows the simple syntax used for feature tagging.

4.3 Updating Feature-to-Code Mappings

Computing the feature-to-code mappings automatically is essential for our approach. The variation control system ECCO[1] stores the entire implementation of the system in the form of artifact trees and maintains mappings between features and the artifacts. Presence conditions determine whether the artifacts will be included in a specific product configuration. A presence condition is a propositional logic formula with feature revisions as literals. ECCO supports the evolution of features over time by considering feature revisions in the computed traces [18]. Our approach uses ECCO's diffing capabilities to automatically update the feature-to-code mappings. ECCO stores implementation artifacts as a generic tree structure. Nodes of the tree are labeled with presence conditions. When committing a new configuration, the tree is updated by automatically computing the presence conditions of the affected artifacts. The tool computes the features' versions and the configurations of each commit and selects commits with relevant tags for checking them out from the repository. In Step 3, the tool incrementally commits all the checked-out versions using their feature configurations to the ECCO repository, thereby automatically updating the feature-to-code mappings.

Features are mapped to arbitrary (parts of) artifacts such as lines of code, statements in source code, model elements, or parts of documents. Features vary in terms of size and scattering. The size represents the number of artifact elements belonging to a particular feature. Features are often realized in multiple artifact locations. Cross-cutting features, in particular, are scattered and realized in multiple contiguous locations.

The user is expected to only tag the actually affected features when annotating relevant commits. When checking out a specific tag using Git, the entire code of all features at this particular commit is included. Therefore, when generating the feature configuration for ECCO the other features are also included automatically to prevent committing a variant with missing features, as this would

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

Table 1: Evaluation scenarios.

	Scenario	Features and their Highest Revisions	#Tagged Commits (#Single Feature Tags)	#Features (#Variants)
S1	Many tags, many features	*Base.19, Traffic.9, Weather.11, I18n.2, Settings.2, Rss.5*	34 (25)	6 (31)
S2	Many tags, combined two dependent features	*Base.19, Combined.15, I18n.2, Settings.2, Rss.5*	35 (25)	5 (15)
S3	Many tags, combined two independent features	*Base.19, Combined.14, Weather.11, I18n.2, Settings.2*	35 (25)	5 (15)
S4	Few tags, many features	*Base.5, Traffic.4, Weather.4, I18n.2, Settings.1, Rss.1*	6 (0)	6 (31)
S5	Highest tags, many features	*Base.24, Traffic.9, Weather.17, I18n.2, Settings.2, Rss.5*	48 (39)	6 (31)
S6	Many tags, few features	*Base.19, V1.16, V2.9*	36 (25)	3 (3)
S7	Few tags, few features	*Base.6, V1.7, V2.3*	8 (0)	3 (3)

result in inconsistent feature-to-code mappings. Therefore, when synthesizing a feature configuration for committing to ECCO, we replace the evolved features with the top-level features' last revision. For example, in the latest version of MiniPaint (commit #6 in Figure 4) the top-level features are *Base, Square, Diamond*, and *Sphere*. The feature configuration for commit #2 on the other hand is *Base.1, Square.1, Diamond.1*, and *Sphere.1*, since the features *Rect, Circle*, and *Oval* no longer exist.

5 EVALUATION

Refactoring a product line by tagging earlier versions of an evolution history is an ambitious goal and has not been investigated to date as far as we know. The goal of our preliminary evaluation is thus to assess if the approach is feasible when applied to a real-world system. In particular, we investigated the following two research questions:

RQ1 – Correctness. To what extent can the mappings of features to code artifacts be computed correctly? We checked the correctness of our approach by experimenting with different refactoring scenarios of the MagicMirror application, covering important combinations of the numbers of features and the numbers of tags. We replayed the annotated version histories of the different scenarios and used our ReSeRVe tool to generate the ECCO repositories. We then composed all possible variants and checked their correctness.

RQ2 – Performance. Can the approach be used in realistic workflows? We measured the performance of each scenario to compute the time needed to generate the feature evolution graph and the ECCO repository for each scenario.

5.1 Data Set

As our dataset, we used the back-end implementation of the MagicMirror application. The system provides capabilities to provide information about the current traffic situation, the current weather and a forecast, a feature to read RSS-feeds from any given source, as well as customization features for the front-end application, including internationalization. We selected the back-end components written in Java for our study. MagicMirror was chosen, because the available plugins for the ECCO variation control system allow parsing its key artifact types, namely Java source code and XML files. The system consists of about 2100 lines of Java code in almost 50 files. Additionally, it also consists of XML files for the project configuration as well as JSON files used for static resources. MagicMirror is modularized with well-managed dependencies between the different modules and can be configured using the SpringBoot framework. For instance, the base of the system can be started without any of the optional modules. Furthermore, one author of the paper is an expert of the system, which facilitated the inspection and analysis of the different automatically composed variants. The evolution history of the MagicMirror version used in this paper contains 100 commits managed in a Git repository. However, the development of the application was already finished before the work on ReSeRVe started, and the developer did not design the system in a feature-oriented fashion.

5.2 Research Method

Regarding RQ1, we performed the following steps: First, we tagged the MagicMirror repository with different levels of granularity. We then used ReSeRVe to replay the annotated version histories and generated ECCO repositories. Finally, we assessed the correctness of the composed configurations, including both intensional and extensional variants. Regarding RQ2, we measured the execution time for creating the feature evolution graph and the creation of the ECCO repository for each scenario. The independent variables, i.e., the characteristics changed to produce different conditions are the number and thus granularity of features as well as the number of tags, i.e., the results for RQ1 and RQ2 depend on the number of tags applied to the history and the granularity of the features. The dependent variables, i.e., the characteristics measured in the experiment, are the level of correctness and the performance of replaying the annotated version histories.

Even when tagging on feature-level the number of used features and the number of tags used to explain changes can vary. To test our approach we thus experimented with seven scenarios covering interesting combinations of inputs summarized in Table 1. We varied our input in terms of the tagging level, i.e., the number of tagged commits, the number of features, and the number of variants. The scenarios were identified by two authors of the paper.

Scenarios including the highest number of tagged commits are assumed to be the best input for our approach, as they provide the most detailed tagging level (S5). Scenarios with many tags provide detailed tagging with most tags only affecting a single feature, e.g., a change is clearly attributed to a particular feature. However, there could still be a few commits affecting multiple features. S7, for instance, has 34 tagged commits out of which 25 affect only a single feature. This is assumed to be an average input for our approach (S1, S2, S3, S6). Scenarios that include few tags only contain high-level tagging involving multiple features, which is assumed to be the worst-case input for our approach (S4, S7). Our preliminary evaluation started with six features, resulting in 31 variants that had to be checked and analyzed. The mandatory feature *Base* was present in every variant. We then reduced the number of features by merging selected features. For instance, scenario S2 was created by combining the dependent features *Traffic* and *Weather*, while scenario S3 was created by combining the independent features *Traffic* and *Rss*.

For evaluation, we checked out different variants from the generated repository. To generate the Java code from ECCO we used a Java plugin developed by Hinterreiter et al. [13]. We used ECCO's default plugins for other artifact types. This means that the XML and JSON files were parsed using the default plugins provided by ECCO. Each variant was then checked for correctness.

We distinguished five correctness levels (CLs) for the automatically composed variants:

CL5 - Correct. In this case, the composed variant compiles and behaves as expected. This was determined by executing the most important test cases of the application.

CL4 - With Surplus. This correctness level means that the system variant compiles and behaves as expected, but only after removing surplus code. This code is added to the variant by ECCO, if a code artifact is mapped to more than one feature, meaning that the input was too ambiguous to automatically compute precise feature mappings.

CL3 - With Runtime Errors. This case is similar to CL4, i.e., the system variant compiles and starts after removing surplus code. However, runtime errors are detected during execution, which require further debugging.

CL2 - With Compilation Errors. This level means that although the checked-out variant contains the required source code, compilation fails even after removing surplus code.

CL1 - Without Feature Separation. In this case source code required for the variant is missing and/or features could not be distinguished based on the input. Hence, the variant contains feature code which should not be included based on its configuration.

6 RESULTS

We discuss our results and findings for RQ1 (correctness) and RQ2 (performance).

6.1 RQ1. Correctness

We checked out all possible 129 variants with the latest feature revisions and examined the automatically composed code. The findings are summarized in Figure 5, which shows the number of variants for the different correctness levels for each of the seven scenarios.

Number of tags vs produced variants. All scenarios result in at least one variant at CL5, i.e., the variant containing all features. This is an extensional case, as the version containing all features was explicitly committed in all scenarios, and ECCO could correctly re-compose these cases, as expected and also discussed in other domains [11]. All other variants of the investigated scenarios are intensional cases that were never committed as such to the repository. Overall, 13/31 of the variants of S1 and S5, 9/15 of the variants of S2, 7/15 of the variants of S3 and 2/3 of the variants of S6 are at CL4 or higher.

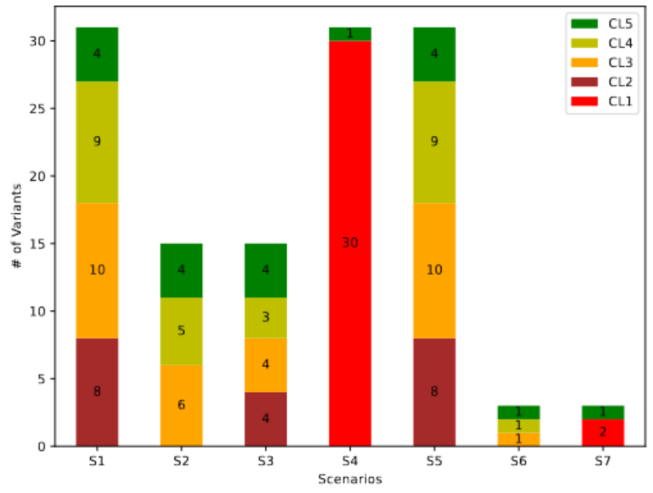

Figure 5: Number of variants for the different correctness levels (CLs) for all seven scenarios. The characteristics of the scenarios are shown in Table 1.

Moreover, none of these scenarios produced variants at CL1. The scenarios with few tags produce variants either at CL1 or CL5, where the CL5 variant is always the one with all features. Since there are so few tags, ECCO is not able to distinguish between the features and always returns the entire code base in these cases. In S4, for example, every variant consists of the same code, resulting in one (extensional) CL5 variant and 30 CL1 variants with no feature separation at all.

The most surprising results were found for scenario S5, for which we added most tags and therefore expected the most accurate variants and many variants at CL4 and CL5. However, there is no difference to S1 when only looking at the correctness levels. When inspecting the different variants of S5, we saw that the produced variants reduced the amount of wrongly added code and S5 is better and worse at the same time compared to S1. For example, the surplus code from S1 was reduced by 41% (from 51 to 31 LOC) for the feature *Traffic* and by 91% (from 857 to 74 LOC) for the feature *Weather*. This shows that increasing the tags (from 34 to 48) while reducing the number of tags affecting multiple features increases the quality of the feature-to-code mappings. On the other hand, the required code for the feature *Traffic* in variants missing the *Weather* feature was reduced by 4% (from 161 to 154 LOC), meaning that the already broken code got even worse. This also shows that tags affecting multiple features negatively influenced the quality of the feature-to-code mappings. However, these changes had no effect on the correctness level, since the following three main problems still persisted for those variants:

Variants with surplus code. ReSeRVe in some cases generates surplus code not needed for a variant, which results in compilation errors. This happened when the underlying variation control system ECCO could not distinguish features completely, which is caused by tagged commits referencing multiple features. In our application this happened to the features *Base*, *Traffic* and *Weather* due to some Git commits referring to some of these features together. Therefore, this problem occured in every variant missing either the feature *Weather* or the feature *Traffic*. This is the case for 23/31 of the variants of S1 and S5, 6/15 of the variants of S2 and 11/15 of the variants of S3. However, these errors can simply be fixed by deleting the surplus code as we will discuss later.

Improper feature interaction. In this case, shared code affected by multiple features is missing. This means that complete classes or certain methods have not been generated while the code still compiles without errors. For instance, in our application the configuration file was empty, which led to runtime instead of compilation errors. This was caused by the *Rss* feature, as every variant not including this feature is missing the shared code. This is the case for 19/31 variants of S1 and S5, 5/15 of S2, and 7/15 of S3. Interestingly, the cause of this problem is not a Git commit affecting multiple features in this case because the *Rss* feature is always referenced in a single Git commit.

Lack of information for mapping features. The third problem deals with compilation errors in required source code, i.e., code required for features is composed incorrectly. This happens when the variation control systems cannot correctly distinguish between features, which is caused by commits and tags affecting multiple features. As already mentioned, this happened for the features *Base*, *Traffic*, and *Weather*. This problem only occurs in variants which contain the *Traffic* but not the feature *Weather*, meaning that *Traffic* depends on *Weather*. However, the problem does not exist in variants containing the feature *Weather* but not the feature *Traffic*. This problem was found in 8/31 variants of S1 and S5 as well as 4/15 variants of S3. Due to the dependency of the features *Traffic* and *Weather* we combined them in S2. Contrary to that, we combined the features *Traffic* and *RSS* in S3 because they seemed to be independent of each other. By combining dependent features, the correctness of the variants increased, especially in CL4 (from 3/15 to 5/15) and CL3 (from 4/15 to 6/15). Further, there is not a single variant with CL2 or lower when comparing S3 and S2.

As expected, scenarios with very few tags, such as S4 and S7, provided no useful results, since every variant consisted of exactly the same code, meaning that ECCO could not distinguish the features due to the ambiguous input. Hence, only the variant with all features is at CL5, while all others are at CL1.

6.2 RQ2. Performance

All measurements were performed on a Dell XPS 15 9560 with 16 GB RAM, an Intel Core i7 7700HQ @ 2.8 GHz with 4 physical and 8 logical cores running Windows 10. The results are shown in Figure 6. Computing the feature evolution graph takes only a few milliseconds, even when using the highest number of tags (48), and thus can be neglected. Generating the ECCO repository requires more time. We focus on the core of our approach and thus measured the time it takes for ECCO to process the checked-out versions, but excluded the time it takes to check out the versions from the annotated version history. The results show that the execution time for generating the ECCO repository increases with the number of tags. The execution time also significantly depends on the amount of data that is committed to ECCO. In our scenarios, the average is 42 files per commit. The time ranges from 2 to 15 seconds, which indicates sufficient performance for practical workflows. Similar

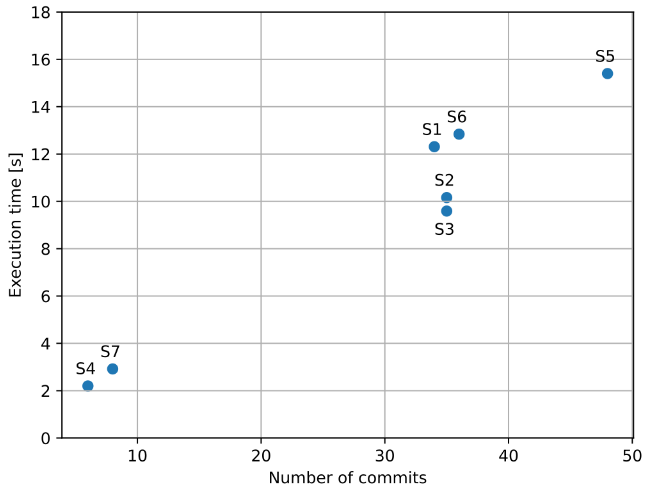

Figure 6: Execution times for computing the ECCO repository for the seven evaluation scenarios.

commit times are reported in evaluations using ECCO for a larger number of features in the context of a large-scale industrial system [13] and in the domain of music engraving [11].

7 DISCUSSION

Impact of Lower Correctness Levels. We provided a detailed discussion of correctness levels for different variants. It is worth discussing the impact of different CLs for practical engineering. While CL5 is obviously the ideal case, also lower levels can often be easily fixed without disrupting the engineering workflow. The reason for lower levels is that ECCO never analyzed a specific variant, and thus problems may occur for certain features that never co-existed before. However, the workflow in ECCO in such cases is to remove the surplus code or to fix the syntax errors in the checked-out variant and then commit the corrected variant, thereby allowing the variation control system to distinguish certain features in future checkouts. Over time, this iterative process improves the mappings of features to code elements, thereby increasing the level of correctness.

Developer Discipline. The discipline of developers when committing features to the original repository has a strong impact on the quality of the results. Obviously, if a tag covers multiple features, they can sometimes not be distinguished by ECCO. In the ideal case, a commit only affects a single feature, which then allows very detailed tagging, thereby increasing correctness. The effect can be clearly seen when looking at the results of scenarios S4 and S7 where all variants, except the ones with all features, are useless because they always contain the entire code.

Iteratively Finding the Right Level of Detail. As can be seen in the results for RQ1, it is not necessary to tag all commits. Given that adding or modifying tags is straightforward and the performance is acceptable, an engineer could start by adding tags for major releases and then add more tags when needed. ReSeRVe would then replay the revised annotated version history to re-create a repository. This can also include splitting and merging features. Afterwards, the required variants can be checked out and tested. This can be repeated until the result matches the expectations. Similarly, the engineer can also experiment with different feature granularities.

Optimizations of Artifact Types. Our approach relies on plugins for the different kinds of artifacts (e.g., Java code, XML files, text files, etc.), which allow the variation control system to correctly compute the feature-to-artifact mappings. Using artifact adapters vastly improves the correctness of the results, as the successful composition of Java code shows. For example, our MagicMirror application also includes JSON and XML files for which we did not use a specific adapter but ECCO's standard text adapter, which certainly leaves some room for improvement.

Usage Scenarios. Our approach can be extended to support incremental replays from Git to update an ECCO repository, which can then be primarily used for product configuration. Another usage scenario is to replace Git with ECCO, which may be less likely, however, due to manifold dependencies on Git in many environments.

Threats to Validity. Our evaluation is based on a small Java system. We identified six features (including the base feature) for our study. However, we still had to manually check each individual composed variant for the selected scenarios, resulting in 129 variants in total. Given this complexity, we argue that the MagicMirror application and repository is acceptable for the purpose of our initial evaluation. It consists of about 2100 lines of Java code and provides an evolution history of 100 Git commits. Additionally, the Java code uses the SpringBoot Framework, meaning that more library functionality is used. Also, our test application consists of static resources as well as project configuration files in different formats like JSON and XML, thereby also demonstrating the feasibility of the approach for artifact types other than code.

The correctness of the results also depends on the correctness of the mappings between the features and the code. In our current implementation, these mappings are computed by the variation control system ECCO, which could influence the results. However, the correctness of the mappings computed during ECCO commits has been evaluated already in earlier research [17].

A threat to internal validity is that we could not evaluate the split command due to limitations of our dataset. As explained, ECCO needs more commits to distinguish features after using the split command and our Git history did not provide sufficient data to experiment with this operation.

8 RELATED WORK

Casquina and Montecchi [6] highlight the need of integrating variability mechanisms and version control systems. They propose an approach for organizing the implementation of SPL features in branches. In order to keep the SPL consistent, any feature change (Git commit) is propagated to other branches, thereby resolving product and down conflicts. The approach was inspired by the work of Hellebrand et al. [12], which follows a similar idea of feature branching for dealing with variability in version control systems. Our approach differs from this research in two points: We organize the source code in the variation control system ECCO to optimize variability management; and we use a lightweight tagging strategy to aid the identification of commonalities and variability compared to these studies.

Variation control systems, such as SuperMod and ECCO, manage revisions and variants of different types of product line artifacts [18]. SuperMod [27] integrates temporal and logical versioning, allowing the development of SPLs in a single-version workspace in a step-by-step manner by using update and commit operations. ECCO [10, 17] can be extended with plugins translating artifacts into its internal tree structure. Plugins have been created for different languages. For instance, Hinterreiter et al. [13] use the variation control system ECCO to manage mappings between features and their implementation in the DSL IEC 61131-3. Similarly, Grünbacher et al. [11] use features managed in ECCO to support the evolution of digital music artifacts encoded in the domain-specific language LilyPond. We selected ECCO for the purpose of this study as we are highly familiar with it and because it provides plugins needed to analyze the code artifacts of the MagicMirror system.

Some studies use version control systems as a source of information for dealing with SPLs. Michelon et al. [22, 23] propose an approach to track feature revisions to their implementation in variants that evolved independently of each other. This is done by analyzing preprocessor annotations in the source code to automatically determine features and their evolution both in terms of space (features being added or removed) and time (feature revisions). Montalvillo et al. [24] propose a solution for reducing SPL development overhead by following the grow-and-prune model for quickly delivering products. They introduce "peering bars" as visual indicators to make engineers aware of features being upgraded in different product branches. This work aids the coordination among SPL engineers. Our refactoring approach migrates SPLs to intensional version control systems, which reduces some potential coordination problems pointed out by these authors.

ReSeRVe uses annotated evolution histories for creating software versions via a variation control system. These annotations can be added manually or automatically. Liebig et al. [14] present an approach to analyze the variability of software systems based on preprocessor annotations, e.g., `#ifdef`. Moreover, they defined a set of metrics for measuring variability. Based on this work, Liebig et al. [15] present a variability-aware refactoring approach for systems using conditional compilation, which aims at preserving the behavior of all variants of a system and interacts with existing refactoring engines.

Loesch and Ploedereder [19, 20] propose an approach for reorganizing software product lines by analyzing product configurations using formal concept analysis. For instance, they determine features not in use anymore, appearing in pairs, or features used in all variants. Based on these categories, their tool-supported approach then suggests features that should be removed, merged, or marked as alternative when used mutually exclusive.

Vierhauser et al. [28] describe an approach for incremental consistency checking on variability models, which also checks the consistency with the underlying code base of the product line. Our approach aims to ensure consistency by replaying a version history and re-creating a feature model on the fly based on annotations instead of fine-grained consistency checking.

Bürdek et al. [5] present an approach that analyzes the differences of feature models by computing the changes as complex edit operations on feature diagrams. The approach can also reason about the semantic impact of diagram changes. This work could complement our approach for product lines with different versions of feature models, as it would assist an engineer in determining feature-level change operations.

Our refactoring approach supports reactive and extractive migration to SPLs [1]. Differently, Bittner et al. [4] proposed a proactive approach in which the traceability of features to source code are obtained during software development via feature trace recording. This is complementary to our work, especially when the proactive adoption of SPLs is an option.

9 CONCLUSIONS AND FUTURE WORK

Extracting information about features and feature-to-code mappings is a problem of high importance in practice, given the large number of legacy systems and software without explicit feature and variant management. This paper presented the ReSeRVe approach that tags an evolution history of an existing product line (e.g., a Git repository) with feature-level operations. ReSeRVe then replays the evolution history and generates a repository of the variation control system ECCO. The approach is useful for both refactoring product lines or migrating existing systems to a product line approach as the tagged commits directly modify the feature model. When using the approach, developers only have to maintain a single repository, as the ECCO repository is generated automatically and can then be used to compose different product variants. While our evaluation is still preliminary it demonstrates the feasibility and potential of the approach. However, more studies are needed for different types of systems and for longer evolution histories to demonstrate the practical applicability and scalability of the approach in more realistic settings.

The ReSeRVe tool can also be combined with a feature mining technique, such as the approach presented by Michelon et al. [23]. In this case a repository could first be tagged automatically. After this, an engineer can further update the generated feature model via additional tags and replay its version history to generate a variation control system. Furthermore, the approach would benefit from integrating code analysis techniques to lift code-level dependencies to the level of features, as shown by Feichtinger et al. [9]. ReSeRVe could also be triggered as part of a build step or in a DevOps environment to automatically assess checked-out selected variants and to report if they compile successfully and if key tests pass.

ACKNOWLEDGMENTS

The research reported in this paper has been partly funded by the Austrian Science Fund (FWF) grant P31989-N31.

REFERENCES

[1] 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.

[2] 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

[3] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. 2015. What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines. In *Proceedings 19th International Software Product Line Conference (SPLC'15)*. ACM, Nashville, USA, 16–25. https://doi.org/10.1145/2791060.2791108

[4] Paul Maximilian Bittner, Alexander Schultheiß, Thomas Thüm, Timo Kehrer, Jeffrey M. Young, and Lukas Linsbauer. 2021. Feature Trace Recording. In *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1007–1020. https://doi.org/10.1145/3468264.3468531

[5] Johannes Bürdek, Timo Kehrer, Malte Lochau, Dennis Reuling, Udo Kelter, and Andy Schürr. 2016. Reasoning about product-line evolution using complex feature model differences. *Automated Software Engineering* 23, 4 (2016), 687–733.

[6] Junior Cupe Casquina and Leonardo Montecchi. 2021. *A Proposal for Organizing Source Code Variability in the Git Version Control System.* Association for Computing Machinery, New York, NY, USA, 82–88. https://doi.org/10.1145/3461001.3471141

[7] 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

[8] Krysztof Czarnecki and Ulrich Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, Boston, MA.

[9] Kevin Feichtinger, Daniel Hinterreiter, Lukas Linsbauer, Herbert Prähofer, and Paul Grünbacher. 2021. Guiding feature model evolution by lifting code-level dependencies. *Journal of Computer Languages* (2021), 101034. https://doi.org/10.1016/j.cola.2021.101034

[10] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *Proceedings 30th IEEE International Conference on Software Maintenance and Evolution* (Victoria, BC, Canada) *((ICSME'14))*. IEEE Computer Society, Washington, DC, USA, 391–400. https://doi.org/10.1109/ICSME.2014.61

[11] Paul Grünbacher, Rudolf Hanl, and Lukas Linsbauer. 2021. Using Music Features for Managing Revisions and Variants in Music Notation Software. In *Proceedings of the International Conference on Technologies for Music Notation and Representation – TENOR'20/21* (Hamburg, Germany), Rama Gottfried, Georg Hajdu, Jacob Sello, Alessandro Anatrini, and John MacCallum (Eds.). Hamburg University for Music and Theater, Hamburg, Germany, 212–220.

[12] Robert Hellebrand, Michael Schulze, and Martin Becker. 2016. A branching model for variability-affected cyber-physical systems. In *3rd International Workshop on Emerging Ideas and Trends in Engineering of Cyber-Physical Systems (EITEC)* (Vienna, Austria). IEEE Computer Society, Washington, DC, USA, 47–52. https://doi.org/10.1109/EITEC.2016.7503696

[13] Daniel Hinterreiter, Lukas Linsbauer, Kevin Feichtinger, Herbert Prähofer, and Paul Grünbacher. 2020. Supporting Feature-Oriented Evolution in Industrial Automation Product Lines. *Concurrent Engineering: Research and Applications* 28 (2020), 265–279. Issue 4. https://doi.org/10.1177/1063293X20958930

[14] 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 *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 105–114.

[15] Jörg Liebig, Andreas Janker, Florian Garbe, Sven Apel, and Christian Lengauer. 2015. Morpheus: Variability-Aware Refactoring in the Wild. In *IEEE/ACM 37th IEEE International Conference on Software Engineering* (Florence, Italy), Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, Washington, DC, USA, 380–391. https://doi.org/10.1109/ICSE.2015.57

[16] Lukas Linsbauer, Stefan Fischer, Gabriela Karoline Michelon, Wesley K. G. Assunção, Paul Grünbacher, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2022. Systematic Software Reuse with Automated Extraction and Composition for Clone-and-Own. In *Handbook of Re-Engineering Software Intensive Systems into Software Product Lines*, Roberto Erick Lopez-Herrejon, Jabier Martinez, Tewfik Ziadi, Mathieu Acher, Wesley K. G. Assunção, and Silvia Regina Vergilio (Eds.). Springer.

[17] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2017. Variability extraction and modeling for product variants. *Software and System Modeling* 16, 4 (2017), 1179–1199.

[18] Lukas Linsbauer, Felix Schwägerl, Thorsten Berger, and Paul Grünbacher. 2021. Concepts of Variation Control Systems. *Journal of Systems and Software* 171 (2021), 110796. https://doi.org/10.1016/j.jss.2020.110796

[19] Felix Loesch and Erhard Ploedereder. 2007. Optimization of Variability in Software Product Lines. In *Proceedings 11th International Software Product Line Conference (SPLC 2007)*. IEEE Computer Society, Washington, DC, USA, 151–162. https://doi.org/10.1109/SPLINE.2007.31

[20] Felix Loesch and Erhard Ploedereder. 2007. Restructuring Variability in Software Product Lines using Concept Analysis of Product Configurations. In *Proceedings 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*. IEEE Computer Society, Washington, DC, USA, 159–170. https://doi.org/10.1109/CSMR.2007.40

[21] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. 2018. Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell. *IEEE Transactions on Software Engineering* 44, 5 (2018), 453–469. https://doi.org/10.1109/TSE.2017.2688333

[22] Gabriela K. Michelon, Wesley K. G. Assunção, David Obermann, Lukas Linsbauer, Paul Grünbacher, and Alexander Egyed. 2021. The Life Cycle of Features in Highly-Configurable Software Systems Evolving in Space and Time. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Chicago, IL, USA). ACM, New York, NY, USA, 14 pages.

[23] Gabriela Karoline Michelon, David Obermann, Lukas Linsbauer, Wesley K. G. Assunção, Paul Grünbacher, and Alexander Egyed. 2020. Locating Feature Revisions in Software Systems Evolving in Space and Time. In *24rd International Systems and Software Product Line Conference* (Montreal, Canada). Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3382025.3414954

[24] Leticia Montalvillo, Oscar Díaz, and Thomas Fogdal. 2018. Reducing Coordination Overhead in SPLs: Peering in on Peers. In *22nd International Systems and Software Product Line Conference - Volume 1* (Gothenburg, Sweden) *(SPLC '18)*. Association for Computing Machinery, New York, NY, USA, 110–120. https://doi.org/10.1145/3233027.3233041

[25] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. 2015. Cloned product variants: from ad-hoc to managed software product lines. *Int. J. Softw. Tools Technol. Transf.* 17, 5 (2015), 627–646.

[26] Sandro Schulze, Jörg Liebig, Janet Siegmund, and Sven Apel. 2013. Does the discipline of preprocessor annotations matter?: a controlled experiment. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*. ACM, New York, NY, USA, 65–74.

[27] Felix Schwägerl, Thomas Buchmann, and Bernhard Westfechtel. 2015. SuperMod — A model-driven tool that combines version control and software product line engineering. In *10th International Joint Conference on Software Technologies (ICSOFT)* (Colmar, France), Vol. 2. IEEE Computer Society, Washington, DC, USA, 1–14.

[28] Michael Vierhauser, Paul Grünbacher, Alexander Egyed, Rick Rabiser, and Wolfgang Heider. 2010. Flexible and scalable consistency checking on product line variability models. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. ACM, New York, NY, USA, 63–72.