Feature-oriented clone and pull operations for distributed development and evolution

Daniel Hinterreiter1 · Lukas Linsbauer2 · Herbert Prähofer3 · Paul Grünbacher1

Accepted: 11 April 2022 / Published online: 25 May 2022
© The Author(s) 2022

Abstract
Software companies frequently customize and extend product lines in multiple projects concurrently to quickly deliver solutions to customers. Engineers use a distributed and feature-oriented development process, commonly supported by version control systems to track implementation-level changes. For instance, feature branches are widely used to add new or modify existing features. However, when merging back features to the product line, the information about feature-to-code mappings is usually lost. Furthermore, the granularity of merging is limited to branches, making it hard to extract and merge selected individual features from one product to another. This paper thus presents feature-oriented clone and pull operations for distributed development, which are implemented in the FORCE² platform. Our evaluation uses variants of the ArgoUML product line to investigate the correctness and performance of our approach. The results show that the feature-oriented operations work with high precision and recall for different cases of feature interactions, also when feature implementations are scattered across many locations in the source code. The performance measurements demonstrate that the operations can be integrated in the typical workflows of engineers.

Keywords Feature-oriented development · Distributed development · Variation control systems · Software product lines
1 Introduction

Software product lines (SPLs) (Pohl et al., 2005) are widely used to provide highly individual solutions to customers. Engineers commonly often follow a feature-oriented approach (Apel et al., 2013) when developing customer-specific systems. They customize and extend product lines concurrently in different projects to quickly deliver solutions, e.g., by creating new features or by selectively reusing and adapting existing features (Hinterreiter et al., 2020b). Supporting such a distributed and feature-oriented development process is challenging: version control systems like Git are widely used to track fine-grained, implementation-level changes to product lines and products. Feature branches are a common mechanism to add new or modify existing features and pull requests allow product line engineers to integrate changes of engineers in a controlled fashion. However, this approach results in two problems (Linsbauer et al., 2021): (i) the mapping of features to code is commonly lost after merging a feature branch. As a consequence, the merged features become mandatory from a product-line perspective and cannot be used to systematically create variants. (ii) In current version control systems, the granularity of integration is limited to branches, i.e., there is no support for merging only selected features from a branch.

This article builds on our earlier work on variation control systems (Linsbauer et al., 2017b, 2021), feature-to-artifact mappings (Feichtinger et al., 2019), and workflows for feature-oriented development (Hinterreiter et al., 2018, 2020b), which already presented feature-oriented commit and checkout operations. We now present feature-oriented operations for distributed feature-oriented development. In particular, we realized and evaluated feature-oriented clone and pull operations for our FORCE² platform, including a thorough evaluation based on a publicly available data set. Specifically, the clone operation allows to create a new product line based on an existing one by including only the features needed for a specific development task. In distinction to existing version control systems (such as Git), it thus composes a system variant with only the features needed for a specific purpose. The pull operation allows transferring feature implementations between individual products or product lines, thus supporting feature-level code reuse. For example, the pull operation can be used to transfer a feature from a cloned product line back to its origin product line to make it generally available. In distinction to Git, this means that also fine-grained variants can be handled, i.e., variability is preserved by managing traces to the artifacts implementing the different product line features.

Specifically, our paper provides the following research contributions: (i) feature-oriented operations for clone and pull to support distributed feature-oriented workflows for software product line engineering; (ii) an implementation of the operations in the variation control system ECCO (Linsbauer et al., 2022), together with an integration in the FORCE² (Hinterreiter et al., 2020b) development environment; (iii) an evaluation based on the ArgoUML product line to investigate the correctness and performance of our approach as well as the complexity of the pulled features.

Our article significantly extends an earlier conference publication (Hinterreiter et al., 2021) as follows: We provide a new section discussing the research background on variation control systems and distributed development. We provide a more detailed explanation of the feature-oriented operations and also added a new example to better illustrate the different pull cases. We provide a new section on the implementation of our approach in the FORCE² IDE. We also added a new research question on the feature size and feature scattering of pulls in our experiment. Furthermore, we significantly extended the discussion of related work.
The paper is organized as follows: Sect. 2 provides background research on distributed development and variation control systems. Section 3 presents an illustrative example for showing a distributed feature-oriented development workflow. Section 4 explains the feature-oriented clone and pull operations. Section 5 briefly describes our implementation based on the FORCE² platform and the variation control system ECCO. Section 6 describes our evaluation method. Section 7 presents the results of evaluating the correctness (RQ1) and performance (RQ2) of our approach and present results on feature complexity (RQ3). Section 8 discusses how our work relates to the existing research. Section 9 rounds out the paper with conclusions and an outlook on future work.

2 Background

Our research is based on version control systems and distributed development:

2.1 Version control systems

Version control systems facilitate managing sequential versions (revisions) and concurrent versions (variants) of software systems (Conradi & Westfechtel, 1998). Widely used version control systems such as Git or Subversion maintain revisions of files, and they support variants via branching and forking. However, they lack support for managing variants at the level of features and allow only coarse-grained variants of whole systems. Developers thus need to additionally employ custom configuration mechanisms (such as pre-processors), resulting in undesirable context switches between tools and mechanisms.

Variation control systems such as ECCO (Linsbauer et al., 2017a, 2021) or SuperMod (Schwägerl & Westfechtel, 2019) address this problem by providing capabilities to uniformly handle revisions and variants. In particular, they allow to decompose a software system into finer-grained variable entities, i.e., features, to manage variability. A recent survey comparing variation control systems identified commonalities regarding their supported workflows and operations (Linsbauer et al., 2021). In particular, variation control systems provide operations to commit new or changed artifacts implementing features to the repository (internalization) and to checkout artifacts from the repository (externalization). As pointed out, these operations are supported at the level of features, i.e., feature-based configurations are used instead of revision numbers or hash values common in existing version control systems.

Using such mechanisms, variation control systems thereby also support intensional versioning, i.e., retrieving and composing new variants that were never explicitly committed to the repository before. This goes beyond the extensional versioning support of version control systems such as Git or SVN, which are limited to retrieving only previously committed versions. In particular, the commit operation of variation control systems assumes an internalization expression specifying the list of features to be committed to the internal representation, i.e., the repository contents not visible to the developer. The checkout operation then expects an externalization expression defining the list of features to be composed in the external representation, i.e., the workspace the developer can interact with and modify.
The approach presented in this article builds on the variation control system ECCO (Fischer et al., 2014; Linsbauer et al., 2017b). ECCO automatically computes presence conditions for implementation-level artifacts based on the provided internalization expression and adds or refines the conditions when necessary. ECCO assumes full configurations as internalization or externalization expressions. This allows developers to compose concrete product variants they need to work with and frees them from manually managing exact presence conditions for artifacts or other variants not in the workspace when adding, modifying, or deleting code. Changes are always made to a concrete product variant and propagated to other affected product variants as needed. In doing so, ECCO also tracks revisions of individual features. Whenever a feature is modified, which is indicated by a marker in the provided configuration, a new revision is created for the feature.

While feature-oriented commit and checkout operations of variation control systems have been described and evaluated in earlier publications (Fischer et al., 2014; Linsbauer et al., 2017b; Hinterreiter et al., 2020b; Grünbacher et al., 2021), the contributions of this paper are distributed operations for feature-oriented software development, possibly involving multiple repositories.

### 2.2 Distributed development

An important aspect of version control systems is their support for collaboration and distributed development. Specifically, fork and pull mechanisms are used by teams collaborating in the development of customer-specific products (Zhou et al., 2019). The independent development of individual functionality (i.e., features) is essential in distributed development. This becomes evident when looking at popular Git branching models that facilitate so-called feature branches, i.e., temporary branches that live as long as it takes to develop a new feature before merging the branch back to its parent. As explained above, a downside of this approach is that Git supports only coarse-grained variations at the level of branches but lacks variations at the level of features. Specifically, a clone or fork created with a version control system such as Git always contains the entire code base, i.e., the implementation artifacts of all features of the product line. It is not possible to include only the desired features when creating a clone. This also means that the mapping to features is lost in Git when merging back a feature branch.

In summary, while variation control systems allow tracking changes at the level of features, they lack support for collaboration and feature-aware operations for distributed development of product lines. Variation control systems on the other hand need to evolve toward distributed platforms for development and evolution (Hinterreiter et al., 2018). This article thus presents distributed operations for variation control systems and realizes them in FORCE2, a feature-oriented development platform and tool environment built on top of the variation control system ECCO.

---

1 The source code of ECCO is available on GitHub at https://github.com/jku-isse/ecco
2 http://nvie.com/posts/a-successful-git-branching-model/
We illustrate typical workflows for distributed feature-oriented development based on the ArgoUML (Ramirez et al., 2011) product variants from the Martinez et al. (2018) feature location benchmark.

Figure 1 presents an overview. The scenario starts with the ArgoUML product line ORIGIN containing the mandatory features Base, Diagrams, Class and the optional features Activity and Logging. The features Base, Diagrams and Class represent the common code parts in ArgoUML. The cross-cutting feature Logging interacts with the feature Activity, i.e., some code is only present if both features are enabled.

In our scenario, an engineer decides to start a new development task based on the ORIGIN product line using the feature-based clone operation. He chooses to include only the mandatory features and the feature Logging from the ORIGIN product line, i.e., thereby excluding the optional feature Activity. The result of the clone operation is the product line CLONE A containing the features Base, Diagrams, Class, and Logging.

In CLONE A the engineer develops a new feature Sequence for modeling sequence diagrams. In step 2, the optional feature Sequence is added to the feature model. Then, he performs a checkout operation to create a product variant P1 including the source code.

3 Distributed feature-oriented workflow

Fig. 1 A scenario illustrating the feature-oriented clone and pull operations in a distributed workflow
of all mandatory and the new optional feature Sequence. However, at this point no source code yet exists for the new feature Sequence. After developing the code of the feature in step 4, the engineer uses the commit operation to add the new source code to the repository, thereby also automatically mapping the new code to the feature Sequence.

The engineer then decides to also develop the code needed for the interaction of the features Sequence and Logging. If two features interact, one feature modifies or influences another feature in defining overall system behavior (Zave, 1993). This means in our context that code needs to be available to ensure the joint operation of these interacting features. For example, the code can be added by checking out and composing a product containing the features Sequence and Logging, adding the code, and then committing the changes (steps 6 to 8). The newly added feature interaction code is then mapped to both features.

Independently of this development task, another engineer derives a product line variant CLONE B from ORIGIN (step 9). She includes the feature Activity but decides to exclude the feature Logging. The engineer then develops the feature Cognitive and commits it to the repository after completion (steps 10-13). The feature Cognitive is cross-cutting; thus, some codes must be added to implement the interaction of the features Activity and Cognitive (steps 14–16).

At this point, the engineer becomes aware of the feature Sequence in CLONE A, which she finds useful for her own task. Therefore, in step 17 she performs a pull operation to retrieve this feature together with the source code mapped to it from product line CLONE A. She then commits these changes to the local repository. However, as the product line CLONE A not yet includes the feature Cognitive, she needs to add code to her implementation ensuring the successful interaction of the features Sequence and Cognitive. In step 18, she thus checks out a product variant containing both features, develops the required interaction code (step 19), and commits it to the repository (step 20).

Finally, the engineer maintaining the product line ORIGIN decides to integrate all features developed in the product line CLONE B. For that purpose, he uses the pull operation to retrieve the features Sequence, Activity and Cognitive (step 21). By pulling these three features, he also receives the code needed for the interaction of the features Sequence, Activity and Cognitive. In this case, the code handling the interaction of the features Sequence and Logging is still missing, as the product line CLONE B did not yet contain Logging. However, the product line CLONE A (for which the feature Sequence was originally developed) contains the feature Logging and thus also the interaction code. Therefore, the engineers fetch the feature interaction code for Sequence and Logging by pulling it from product line CLONE A (step 22). In this scenario, the feature interaction code of Cognitive and Logging might still be missing, in which case the engineer of the product line ORIGIN would need to develop and commit it in the same way as described already for other interactions.

### 4 Feature-oriented clone and pull operations

The scenario in Sect. 3 demonstrates the need for clone and pull operations in distributed development. These operations need to be feature-oriented and allow engineers to clone or pull a single feature, a set of features, or a (partial) configuration as needed for their development tasks. We now describe the two feature-oriented operations clone and pull.
Our approach relies on a product line platform \( PL \). Formally, we define a product line platform \( PL \) as the tuple

\[
PL = (F, FM, M, FR)
\]

comprising the features \( F \) of the product line, constraints among features in the form of a feature model \( FM \), a set of fragments of implementation artifacts \( FR \) (e.g., lines of source code), as well as mappings \( M \) between fragments and features in the form of presence conditions (Hinterreiter et al., 2020b). The presence conditions are Boolean formulas in disjunctive normal form (DNF) with features \( f \in F \) as literals. Thus, a DNF formula is a disjunction of clauses, and every clause is a conjunction of features and can be interpreted as a feature interaction, i.e., fragments mapping to such a condition implement the interaction of the involved features. A mapping \( m \in M \) then is a pair \((c, FR')\) of a presence condition \( c \) and a set of fragments \( FR' \subseteq FR \).

An example of a presence condition in ArgoUML is \((Activity \land \neg Cognitive) \lor \neg Cognitive\). Artifact fragments with this mapping are included in any product variant containing the feature \( Activity \) but not the feature \( Cognitive \). The corresponding mapping \( m_2 \in M \) would be the pair \((Activity \land \neg Cognitive, \{fr_3, fr_4, \ldots\})\). Many presence conditions in a product line, however, consist only of a single variable, e.g., \( Activity \), meaning that the associated code fragment belongs to a specific feature.

### 4.1 Feature-oriented clone

The feature-oriented clone operation is used to derive a new product line from an existing one. While the clone operation in Git copies the entire repository (i.e., product line), our feature-oriented clone allows to exclude optional features, thus creating a more specific product line platform needed for a specific development task. It also adapts and pre-configures the feature model, i.e., deselected features are removed from the model, while selected features may become mandatory or remain variable. The cloned platform then serves as the foundation for further development, i.e., it allows to add new features or modify existing ones. If an engineer realizes that further features are required from the original platform or features implemented in other platforms, they can also be fetched using pull (cf. Sect. 4.2).

More formally, the clone operation takes as input a product line \( PL_{\text{origin}} \) and a set of features \( F \) and produces as output a product line \( PL_{\text{clone}} \) containing only the selected features in \( F \).

\[
PL_{\text{clone}} = \text{clone}(PL_{\text{origin}}, F \subseteq F_{\text{origin}}) = \text{subset}(PL_{\text{origin}}, \lnot \{ f \mid f \notin F \})
\]

The clone operation is based on the subset operation \( PL' = \text{subset}(PL, C) \), which takes as input a product line \( PL \) and a partial feature configuration \( C \) and produces as output another product line \( PL' \) that is a subset of the original product line \( PL \) (i.e., \( PL' \subseteq PL \)). This means that the features and supported product variants of \( PL' \) are a subset of the features and supported product variants of \( PL \).
A feature configuration is a set of positive or negative features. A partial feature configuration is a feature configuration in which some features are selected or deselected (have a value true or false assigned), while others remain variable (i.e., undecided).

All features with a value assigned in \( C \) are no longer variable in the target product line, i.e., they are either removed (value false assigned) or become mandatory (value true assigned), in which case they could optionally be merged with their parent feature. The latter may be useful, for example, if a company wants to restrict access or limit knowledge about unlicensed features or trade secrets.

The relation between the set of features \( F \) provided by the user to the clone operation and the partial configuration \( C \) computed as input for the subset operation is such that the features \( f \in F \) have no value assigned in the partial configuration \( C \) and thus remain as they are (i.e., optional or mandatory) and the features \( f \notin F \) are negated in \( C \) and are thus not part of the cloned product line.

Moreover, the subset of features \( F \) selected for the clone operation, and thus the partial feature configuration \( C \), must be consistent, i.e., must fulfill the feature model constraints, in the sense that all the feature constraints are fulfilled. For example, if a feature from an XOR-group is set to true to become mandatory, all other features in the XOR-group can no longer be selected and are set to false. However, if a feature in an XOR-group is selected but kept variable, other features in the group can still be set as variable or be deselected.

**Artifact Fragments and Mappings** The cloned subset product line \( PL' \) contains only the mappings and related fragments of artifacts of the original product line whose presence conditions are still satisfiable given the partial feature configuration \( C \), i.e., only code fragments with satisfiable presence conditions become part of the cloned platform. In particular, features set to true or false may render presence conditions unsatisfiable. However, features that are still variable remain as feature variables in the DNF clauses. For example, presence conditions with a single clause, e.g., \( Activity \land \neg Cognitive \), or just a single feature, e.g., \( Activity \), become unsatisfiable as soon as the feature \( Activity \) is deselected. Finally, the resulting presence conditions are trimmed such that DNF clauses, which are no longer satisfiable, are removed from the presence conditions.

For example, consider a product line \( PL \) with optional features \( Activity, Sequence, Logging, \) and \( Cognitive \) (cf. Fig. 1). The product line should contain a mapping \( m_1 \) with condition \( Activity \), a mapping \( m_2 \) with condition \( Activity \land \neg Cognitive \) and a mapping \( m_3 \) with condition \( (Activity \land Logging) \lor (Sequence \land Logging) \). When creating a clone by selecting the features \( Activity \) and \( Sequence \) but excluding the features \( Logging \) and \( Cognitive \), i.e., \( \text{clone}(PL, \{\neg Logging, \neg Cognitive\}) \), the condition \( (Activity \land Logging) \lor (Sequence \land Logging) \) is no longer satisfiable and the mapping \( m_3 \) and its associated code fragments are not included. The condition \( Activity \lor \neg Cognitive \) of mapping \( m_2 \) on the other hand becomes \( Activity \), which is still satisfiable, meaning that its code fragments will be included. The condition \( Activity \) of \( m_1 \) (which is not modified) is also satisfiable, and \( m_1 \) and its code fragments are therefore also included.

After this step, multiple mappings may still have the same condition. In the above example, the mapping \( m_2 \) with condition \( Base \land \neg Cognitive \) that was trimmed to \( Base \) is identical to the condition of mapping \( m_1 \) that only had condition \( Base \) to begin with. Consequently, \( m_2 \) can be merged into \( m_1 \).
4.2 Feature-oriented pull

The feature-oriented pull operation allows transferring one or more features between related product line platforms. In contrast to the pull operation of Git, which always pulls an entire platform, the feature-oriented operation only fetches the selected set of features and associated artifacts. Besides performance benefits, this approach allows getting specific features from a product line without the need to bother with other features, for which different versions may already exist in the target platform.

More formally, the pull operation takes as input a product line $PL_{origin}$, a set of features $F \subseteq F_{origin}$, and a product line $PL_{target}$ and produces as output a product line $PL'_{target}$ containing only the selected features in $F$.

$$PL'_{target} = \text{pull}(PL_{origin}, F \subseteq F_{origin}, PL_{target}) = \text{merge}(\text{subset}(PL_{origin}, \{\neg f \mid f \notin F \land f \notin F_{target}\}), PL_{target})$$

The pull operation is based on the previously explained subset operation and followed by a merge operation. Thus, the pull operation is also based on a partial feature configuration $C$ representing the set of features to retrieve. However, the partial configuration is built in a slightly different manner based on the set of features $F \subseteq F_{origin}$ the user selects for pulling. The relation between the feature $f \in F$ provided by the user and the partial feature configuration $C$ used for the subset operation is the following. When a user specifies a feature $f \in F$ for pulling:

(i) the specified feature $f$ remains variable (i.e., is NOT set to true as it would then become mandatory in the target platform),
(ii) all interactions of the selected feature with other features already in the target platform will also be pulled, and
(iii) all other features are deselected, i.e., set to false, in the partial feature configuration used in the subset operation.

Note that case (ii) guarantees that also feature interactions of the selected feature, which are needed in the target platform, are pulled. The approach naturally expands when a number of features are selected for pulling. The features $\overline{F} = \{\neg f \mid f \notin F \land f \notin F_{target}\}$ that are not pulled are not transferred, i.e., set to false in the partial feature configuration for the subset product line being transferred. A feature or set of features can be pulled from an origin platform if the required constraints can be fulfilled and all required artifact fragments (e.g., source code elements) are either already contained in the target platform or part of the artifact fragments to be pulled from the origin platform. The subset operation is used to only transfer the subset of features from the origin platform that is requested for the target platform instead of the entire product line.

The merge operation then integrates the transferred subset product line into the target product line. Specifically, the $PL'_{target} = \text{merge}(PL'_{pull}, PL_{target})$ receives two product lines $PL'_{pull}$ and $PL_{target}$ as input and merges them to another product line $PL'_{target}$ such that $PL'_{target}$ is the union of the product lines $PL'_{pull}$ and $PL_{target}$.

Merging two product lines means merging the feature models as well as the artifact fragments and mappings. Feature models are merged by making as few changes as possible in the target platform and leaving the fundamental decisions about the intended structure of the feature model deliberately to the engineer. When a pulled feature is already contained in the target platform, the pulled feature is added as a new revision. When a feature is already contained in the target platform, but in different positions in the feature models, the engineer is notified and the feature remains on its original position in the target model. In case the pulled feature does not yet exist in the target feature model, it is inserted at the
feature which is the next common ancestor in both models. In both cases, it is up to the engineer to make the final decision about the appropriate position and possibly update the constraints in the target feature model.

Artifact Fragments and Mappings The pull operation then adds the set of mappings and related artifact fragments to the target platform. Adding presence conditions and code fragments to the repository is followed by an automatic consolidation step checking if the presence conditions of mappings overlap and then splitting and/or merging mappings accordingly. This step is necessary because mappings are potentially of different granularity. For example, in a product line PL\textsubscript{origin} two mandatory features may be mapped to the same artifact fragments if they could never be separated in earlier commits. However, this may not be the case in another product line PL\textsubscript{pull} (cloned from PL\textsubscript{origin} at some point), if one of the two features became optional, thereby splitting up the artifact fragments accordingly.

Complete or Incomplete Pulls The pull operation merges feature code from an origin platform with the target platform. The state of the platform repository depends on whether the pull was complete or incomplete. This again depends if the features of the target platform are a subset of the origin platform, and if the pull results in feature interactions (Zave, 1993), i.e., code handling the joint behavior of certain features is missing. Figure 2 illustrates four basic cases using a simple example. In a complete pull (cases I and II), no further interaction of an engineer is required to generate a semantically and syntactically correct product line variant. In an incomplete pull (cases III and IV) certain artifact fragments
and their mappings are missing. This usually happens if the target platform repository contains features not contained in the origin repository.

I. **Pulling features from a superset.** A pull is always complete if the origin platform contains a superset of the features and their revisions in the target platform. This means that the origin platform \((F_1, F_2, F_3)\) contains all features and feature revisions also contained in the target platform \((F_1)\), thus also the glue code handling all possible feature interactions. The pull \((F_2, F_3)\) will thus succeed. Recall, that case (ii) in building the partial feature configuration for the pull causes this feature interaction code to also be transferred to the target platform, thus making it complete.

II. **Pulling independent features from a non-superset.** A pull can also be complete if the origin platform \((F_1, F_2, F_3)\) contains a non-superset of the features and their revisions in the target platform \((F_0)\). However, there must be no interaction of the pulled features with the features in the target repository for pull \((F_2, F_3)\) to succeed.

III. **Pulling interacting features from non-superset.** This case is identical to pull case II in terms of the two repositories; however, the interaction code needed for the combined use of features is missing, i.e., pull \((F_2, F_3)\) will result in missing code. There are different ways to address this problem as illustrated from a developer perspective in Sect. 3. An engineer can check out a variant containing the feature combinations and then implement the missing glue code, which enables these feature to work together properly, in our example \((F_0 & F_2)\) and \((F_0 & F_3)\). Afterward, using the commit operation, the glue code can be incorporated into the repository.

IV. **Pulling features from multiple non-supersets.** This case is identical to pull cases II and III in terms of the two repositories. As in case III the interaction code needed for the combined use of features is missing. However, as shown in the scenario in Sect. 3 the engineer in this case does not develop new code but pulls the missing feature interaction code from another repository already containing both features and thus their corresponding glue code, in the example \((F_0 & F_2)\) and \((F_0 & F_3)\).
5 Implementation

The feature-oriented clone and pull operations have been implemented in Java as a plugin for the Eclipse IDE as part of the FORCE$^2$ approach. Figure 3 shows the architecture of the tool environment consisting of the following components:

The code model represents the code artifacts of various kinds in an integrated abstract syntax tree implemented based on the Abstract Syntax Tree Metamodeling (ASTM) standard (OMG, 2011; Grimmer et al., 2016), which provides a specification for code models written in different languages. The reader/writer framework allows parsing and persisting different types of product line artifacts. By providing reader and writer plugins, the system can be customized to handle different kinds of artifacts, e.g., different programming languages. For example, we implemented reader and writer plugins for the Java language based on the JavaParser$^3$ framework and reader and writer plugins for the various configuration files needed for the ArgoUML benchmark study. The feature-oriented and distributed variation control system ECCO is used for computing and maintaining the feature-to-code mappings and product line artifacts. ECCO’s API provides access to the low-level subset and merge operations as well as the high-level distributed clone and pull operations described in the previous section. The FORCE$^2$ core component implements support for temporal feature modeling (Hinterreiter et al., 2019) as well as the core operations for distributed development. It relies on the feature-to-code mappings from the ECCO system and operates on the code model. Further, the code dependency analysis component is capable of analyzing and representing all code-level control and data dependencies in a system, also considering its variability (Angerer et al., 2014, 2019). The FORCE$^2$ feature dependency analysis component allows analyzing and representing dependencies

---

3 https://javaparser.org/
at the levels of features (Feichtinger et al., 2019). It relies on the code-level dependencies and the feature-to-code mappings encoded as ECCO presence conditions. Finally, the Eclipse plugin provides a user interface for performing the various operations supported by FORCE². The FORCE² Eclipse plugin provides the FORCE² functions through several advanced UI components to the user. A project explorer allows managing multiple product lines and their fragments. The explorer also serves as a front end for triggering the high-level operations commit, checkout, clone, and pull (cf. Fig. 4). With a temporal feature modeling editor (Hinterreiter et al., 2019), feature models can be displayed and edited. It also supports tracking and analyzing feature model evolution by showing differences between versions of a feature model. With a source viewer one can inspect artifacts at the source code level. The viewer is also capable of showing feature mappings, i.e., the lines mapped to a selected feature, and showing differences of versions, e.g., the difference of a code artifact before and after a pull operation. It allows a developer to inspect the results of her actions, e.g., pulls, at the level of code. Further, there are views for displaying the results of code and feature-to-feature dependency analyses and various wizards for supporting the distributed operations.

Figure 4 illustrates the tool support for carrying out the clone and pull operations for the example from Fig. 1. The upper half shows the feature model of the origin and the dialog for creating a clone CLONE A (cf. step 1 in Fig. 1): the optional feature Logging is selected, while the feature Activity is deselected. Below, on the left side, the resulting feature model of CLONE A is shown, already with feature Sequence added (after step 8 in Fig. 1). CLONE B is created in the same manner. The dialog in the lower middle is then used for pulling the feature Sequence to the model CLONE B (cf. step 17 in Fig. 1). Finally, the screenshot on the right shows the resulting feature model of CLONE B.

6 Evaluation

Our evaluation pursues three research questions regarding the correctness and performance of our approach, as well as the complexity of the pulled features.

Correctness (RQ1) – Do our feature-oriented operations provide correct results? Fig. 1 makes clear that a high-level of correctness of the feature-oriented clone and pull operations is essential to support distributed workflows in product line engineering. In particular, this means that the feature-oriented operations need to work with high precision and recall for different cases of feature interactions. In Sect. 4, we showed that both clone and pull rely on the subset operation. The clone operation is realized by pulling the selected features into an empty repository. Our evaluation thus investigates the correctness of the pull operation in different scenarios and for different cases of feature interactions. As discussed in Sect. 4 a pull is complete if the origin platform contains a superset of the features in the target platform. In this case, no further interaction of an engineer is required to generate a semantically and syntactically correct product line variant. Otherwise, the pull is regarded as incomplete and closer inspections are required to assess correctness. Section 6.2. summarizes the different cases we explored for the pull operation.

Performance (RQ2) – Does the execution time of the feature-oriented operations allow their use in engineering workflows? Pulling features from one product line to another is a frequent activity, as shown in the scenario in Fig. 1. The operations can only be integrated in everyday workflows of engineers if the performance is sufficient for feature with
different properties. We thus evaluated the run-time performance of the pull operation for different numbers of feature-to-code mappings and artifact fragments.

**Feature Complexity (RQ3)** – How complex are the pulled features in terms of their size and scattering? Features may differ significantly regarding the size of the artifacts used to implement them. Features may further be realized across many locations in the implementation. The approach should work with features of different size and scattering to support realistic workflows in distributed product line engineering. We thus investigated the complexity of the features in our experiment using the metrics feature size and feature scattering. It has been shown that these metrics allow to estimate the effort required to manually extract and integrate a feature into another product variant (Hinterreiter et al., 2020a).

### 6.1 Method

We used the well-known ArgoUML system for evaluating our approach and adopted the ArgoUML benchmark suite provided for a feature location challenge at SPLC 2018 (Martinez et al., 2018) for our experiment.

Figure 5 gives an overview of the research method we used for our evaluation. The ArgoUML benchmark suite allowed us to generate all 256 feasible variants of ArgoUML based on its eight optional features. We first created FORCE² platforms and repositories for all ArgoUML variants based on a script containing a series of commit operations. The order of the commits mimics the evolution from a small variant containing just the base feature to larger variants containing all of the available features. Note that the correctness of the commit operation was evaluated already in earlier work (Linsbauer et al., 2017b; Michelon et al., 2019; Grünbacher et al., 2021).

The evaluation then continued with both a manual procedure investigating RQ1 and an automated procedure assessing both RQ1 and RQ2. We also retrieved the data for RQ3 from the ArgoUML variant repositories. Our first research question investigates whether the feature-to-artifact mappings of the platform repository created via pull operations are equal to the mappings created via the commit operations. For the manual part, we executed 15 scenarios of pulling feature sets of different sizes to cover different cases of incomplete pulls (cf. Sect. 6.2). We manually inspected and compared these cases based on our ArgoUML baseline. For the automated part, we developed a script creating FORCE² platforms and repositories based on existing ones and then incrementally extended them using pull operations. In particular, for this part of the evaluation we pulled features from a FORCE² platform containing a superset of the features of the platform (pull case I in Sect. 6.2). This process was done for all FORCE² platforms, i.e., all ArgoUML variants.
Specifically, our script starts with a repository only containing the mandatory features (Base, Diagrams, Class) of ArgoUML. Hence, with respect to this basic platform, all other ArgoUML variants represent supersets, meaning that pulling features would be possible from any platform. However, the script incrementally increases the number of platform features with one new feature per pull. Hence, we pull a single optional feature from a repository containing exactly this feature in addition to the common features in the platforms. We repeat this process for each variant containing a single optional feature, always starting with a repository only containing the base features. After the first iteration, the script starts with the repositories containing the newly pulled optional feature, copies them and continues with pulling another optional feature from a superset platform. The resulting projects can then be compared to their baseline FORCE project variants to answer RQ1.

We use precision and recall to measure the correctness of the automated part: Precision is defined as the number of matching feature mappings contained in both the origin and the target repository divided by the overall number of mappings in the target repository. We determine across all pulls the proportion of correctly pulled relevant artifacts.

\[
\text{precision} = \frac{|m_{\text{base}} \cap m_{\text{target}}|}{|m_{\text{target}}|} \tag{4}
\]

Recall is defined as the number of matching feature mappings contained in both the origin and the target repository divided by the overall number of mappings in the origin repository. We determined across all pulls how many relevant artifact traces are pulled, i.e., are all artifact traces contained in the base also contained in the target?

\[
\text{recall} = \frac{|m_{\text{base}} \cap m_{\text{target}}|}{|m_{\text{base}}|} \tag{5}
\]

Regarding RQ3, we computed the feature size and feature scattering for all pulls from the repositories. We used these metrics based on definitions in earlier work (Hinterreiter et al., 2020a) to estimate the maintenance effort of the pulled features:

Feature size (FSi) is measured by the magnitude of artifacts mapped to a particular feature (e.g., the number of program elements in source code or the number of data elements in XML files).

Feature scattering (FSc) is determined by the number of contiguous locations of a feature’s implementation, often across multiple artifacts. For example, if a feature is mapped to all source files within one directory, the directory represents the location of the implementation and the scattering is 1. However, scattering is 5 if the feature is mapped to five independent source files.

6.2 Pull cases

In case of tightly coupled features, we can expect interactions between them. For instance, cross-cutting features often require certain interaction code to work correctly with other features. Examples are the features Logging and Cognitive in the ArgoUML case study. No interactions are expected if features do not exchange data or do functionally not rely on each other. However, even in such cases interactions may be discovered in the code, e.g., related to the configuration, initialization or declaration of features, meaning that source code often depends on the presence of certain features. If such interactions cannot be pulled from the origin repository, glue code may need to be developed. However, often the
feature interaction code is already contained in the origin platform repository as we will show. As already mentioned in Sect. 4.2 pulls can have different outcomes with respect to their completeness, and our evaluation also distinguishes these four cases:

I. **Pulling features from a superset.** In this case, a feature (or set of features) is pulled from an origin platform repository containing a superset of the features in the target platform. This means that the pull can be performed completely as all potential interactions are already included in the target platform. We cover this case with the automated evaluation procedure (cf. Sect. 6.1), i.e., we create repositories by incrementally pulling features from superset repositories and compare the resulting repository with the one created with the commit operations.

II. **Pulling independent features from a non-superset.** Pulling features from an origin that contains features which are not a superset of the features in the target platform will often result in missing feature interaction code. In this case, the pull can still succeed completely if the pulled features are truly independent from any features in the target platform. We did not assess this pull case in our evaluation as there are no truly independent features in the ArgoUML data set.

III. **Pulling interacting features from non-superset.** If a target platform contains features not contained in the origin platform and one or more interacting features are pulled from an origin, the target platform will miss glue code needed for the joint operation of the existing features. In order to solve this problem, the engineer needs to implement the glue code ensuring the correct interaction of those features. There are many possibilities with respect to the kinds of artifacts that might be missing as we will discuss later. To cover this pull case in our evaluation we manually selected eight variants (cf. Table 1) and pulled certain features from an origin to a target platform. When selecting variants we excluded trivial variants (e.g., target repositories with only base features) and included different sizes of repositories and pull feature sets. The goal of this evaluation was to verify that the only missing elements in the repository are the ones representing the feature interactions.

IV. **Pulling features from multiple non-supersets.** As just described, pulling cross-cutting features leads to missing interaction code with the existing features in the target platform. However, an alternative to developing the missing interaction code is to pull it from a platform already containing that interaction code. A prerequisite for this operation is that the involved features in the origin and target platform share the same revision. Note that pulling the interaction code also works for different revisions, but requires merging of the revisions to solve potential issues. The features in the ArgoUML case study do not have revisions and thus pulling missing feature interactions works without further involvement from an engineer. For the evaluation of this case we thus selected seven scenarios (cf. Table 2) for which feature interactions exist that require the implementation of glue code or pulling of interactions from other repositories. We reused the selected variants from the previous evaluation step and only excluded one variant with no interaction present. We pulled the missing interactions from another repository containing them and again checked if the repository contains all required elements after the pull. Obviously, directly pulling the features from the more complete repository would be more efficient. However, we wanted to demonstrate that pulling different parts of a feature and feature interactions from different repositories works correctly.
7 Results

We summarize and discuss the main results of our evaluation.

7.1 RQ1: Correctness

I. Pulling features from a superset. The automated part of our evaluation shows very high values for precision (0.987) and recall (0.966). While these numbers are very good, it is interesting to find out what prevented perfect values. Upon closer inspection of selected repositories, we learned that the small problems are not caused by the pull operation itself but are already rooted in the commit operation of the variation control system. Specifically, the diffing and merging algorithms of the ECCO commit operation already introduce these slight imperfections. As a consequence, in some rare cases no presence condition was assigned to artifact mappings. In particular, this happened over the course of committing different variants when for artifact fragments with an initially assigned valid condition a contradicting condition for the supposedly (according to the diffing algorithm) same artifact fragments was found in a later commit. This minor problem can be fixed by improving the comparison and matching algorithm in ECCO or by using coding conventions to avoid problematic program structures (cf. Sect. 7.4).

III. Pulling interacting features from non-superset: The results for the first part of the manual evaluation case are presented in Table 1. The first column Origin lists the features present in the platform from where a feature was pulled. The second column Target shows the features which were already present in the platform. The third column lists the features pulled from origin. The fourth column presents the expected missing mappings. All test scenarios met the expectations with respect to the missing mappings and corresponding artifact fragments. However, as in case I, we discovered some unexpected missing mappings, which are due to imperfection of the matching algorithm or artifact plugin of ECCO. The missing elements are mostly related to initialization and declaration aspects, or user interface elements changing slightly depending on the features present. Again, such problems could be fixed by improving the source code structure or the mapping precision (cf. Sect. 7.4).

IV. Pulling features from multiple non-supersets: Table 2 presents the results of the second part of our manual evaluation. Specifically, the columns Origin 1-3 list the features contained in the different platforms from which features were pulled. The column Target lists the features contained in the target platform before executing the pulls. The three right columns describe the features pulled from the different origin platforms: Pull from 1 presents the features which are originally of interest to be reused in the target platform and pulled from Origin 1 while Pull from 2 and Pull from 3 are the feature combinations pulled to retrieve the missing interactions from Origin 2 and Origin 3, respectively. Again, all results were correct, meaning that after all pulls no artifacts or traces were missing, and no unnecessary elements were transferred.

Overall, regarding RQ1 our evaluation results show very high precision and recall of the feature-oriented pull operation for transferring features and corresponding artifacts between different platforms. The investigation of specific pull cases confirms the expected behaviour for selected cases of interacting features.
7.2 RQ2: Performance

During the automated execution of our pull operation, we measured the run time for the execution of each pull and also logged the number of mappings and artifacts involved. These measurements were executed on a Windows 10 system, with an Intel Core i9-9900K 3.6GHz, 32GB RAM using Hotspot Java VM 1.8 inside an Eclipse IDE.

The results of the run-time measurement of more than 1000 pulls are presented in Fig. 6. The run time lies between about 8 and 14 seconds. Figure 7 shows the number of feature mappings pulled for all pulls. There is a high correlation of the run time and number of mappings involved in the pull operation with a Spearman rank correlation coefficient of 0.86 ($p < 0.01$). We also investigated present the distribution showing the number of artifact fragments pulled from the origin platform for all pulls. Numbers mostly vary between 1000 and 4000 artifact fragments; some outliers have more than

![Fig. 6 Run time for the automated execution of all pulls](image)

| Origin          | Target         | Pulled      | Missing Feature Interaction Code         |
|-----------------|----------------|-------------|------------------------------------------|
| B, UC           | B, ST, cog     | UC          | No Interaction                           |
| B, SQ           | B, ST, cog     | SQ          | SQ-cog                                  |
| B, SQ, cog      | B, ST, UC, log | SQ          | SQ-log                                  |
| B, DP           | B, cog         | DP          | DP-cog                                  |
| B, DP, CL, cog  | B, AC, UC, cog, log | DP, cog | DP-log, CO-log, UC-DP                  |
| B, AC, SQ       | B, cog, log    | AC, SQ      | AC-log, AC-SQ, SQ-log                   |
| B, AC, UC, CL   | B, ST, DP, SQ, log, cog | AC, CL | AC-log, AC-ST, UC-SQ, CL-SQ, CL-log, CL-SQ-log |
| B, log          | B, AC          | log         | AC-log                                  |

*B Base+Class diagram, ST State diagram, SQ Sequence diagram, UC Use Case diagram, CL Collaboration diagram, DP Deployment diagram, AC Activity diagram, log Logging, cog Cognitive
Table 2  Pull Case IV – We checked for the following cases that no code is missing when pulling missing interaction code from multiple repositories

| Origin 1  | Origin 2          | Origin 3          | Target       | Pull from 1 | Pull from 2 | Pull from 3 |
|----------|-------------------|-------------------|--------------|-------------|-------------|-------------|
| B, SQ    | B, SQ, cog        | B, SQ, ST         | B, ST, cog   | SQ          | SQ, cog     | SQ, ST      |
| B, SQ, cog| B, ST, SQ, log    | -                 | B, ST, UC, log| SQ          | SQ, log     | -           |
| B, DP    | B, DP, cog        | -                 | B, cog       | DP          | DP, cog     | -           |
| B, DP, CL, cog| B, DP, UC, log | B, CL, AC, log   | B, AC, UC, cog, log | DP, cog | DP, log | CL, AC, cog |
| B, AC, SQ| B, AC, LC, cog    | B, SQ, log, cog   | B, CL, log   | AC, SQ      | AC, SQ, log, cog | - |
| B, AC, UC, CL| B, AC, SQ, ST, UC, DP, log, cog | - | B, ST, DP, SQ, log, cog | AC, CL | AC, ST, UC, SQ, CL, log | - |
| B, log   | B, AC, log, cog   | -                 | B, AC        | log         | AC, log     | -           |

*B* Base+Class diagram, *ST* State diagram, *SQ* Sequence diagram, *UC* Use Case diagram, *CL* Collaboration diagram, *DP* Deployment diagram, *AC* Activity diagram, *log* Logging, *cog* Cognitive
14000 artifacts. However, the number of artifact fragments did not have a noticeable influence on the runtime of the pull operation.

*Regarding RQ2, our performance evaluation demonstrates that the operations are fast enough to be integrated in the development workflows of engineers. The number of artifact fragments mapped to the pulled feature had no significant impact on performance.*

![Number of feature mappings for all pulls](image1)

![Feature size distributions of eight features transferred in the pulls of the experiment](image2)
7.3 RQ3: Feature complexity

The runtime of the pull operation (RQ2) is not directly related to the maintenance effort of integrating a pulled feature (RQ3). The development effort is difficult to predict in an experiment, but certainly depends on the size and scattering of a pulled feature to be integrated. It has been investigated that features scatter significantly in industrial systems, which increases the maintenance effort (Angerer et al., 2014; Hinterreiter et al., 2020a).

To investigate the complexity of the features in ArgoUML, we measured the feature size and feature scattering for all features pulled in our experiment. Figure 8 uses block diagrams to show the distribution of the feature size of the eight features in the benchmark, while Fig. 9 shows the feature scattering for the same features.

The features LOGGING and COGNITIVE are of particular interest: the LOGGING feature is quite small but has a high feature scattering. Only little code was required to implement the logging functionality, while the code is cross-cutting and has many interactions with other features. On the contrary, the COGNITIVE feature is large due to advanced functions, but at the same time hardly scattered in the code base.

These numbers can help to estimate the effort that would be required to manually transfer features between different variants. Although LOGGING is small it may be hard to transfer due to the high scattering. While COGNITIVE is very large its lower scattering suggests manageable effort for transferring and integrating this feature.

Regarding RQ3, the evaluation of the size and scattering of features shows that the approach also works for large and highly-scattered features, with artifacts in many
locations in the source code. This is important as the manual integration of such features can represent a significant challenges for developers, which can be facilitated with our approach.

7.4 Threats to validity

FORCE2 utilizes ECCO as a variation control system. One might argue that the generated FORCE2 platforms cannot be used as a baseline for checking the correctness of the pull operation. However, existing research (Michelon et al., 2019) already demonstrated that ECCO extracted the location of features, i.e., feature-to-artifact mappings, for ArgoUML variants with high precision. We did not evaluate and investigate the ECCO commit and checkout operations as part of this paper, as positive evaluation results are already reported in existing research (Michelon et al., 2019; Fischer et al., 2014; Linsbauer et al., 2016, 2017b; Hinterreiter et al., 2020b; Grünbacher et al, 2021).

Additionally, there might be a bias due to using ArgoUML, which consists primarily of Java source code. We did not investigate the potential influence of other artifact types. Furthermore, the ArgoUML software product line was originally extracted from ArgoUML for a feature location benchmark (Martinez et al., 2018). ArgoUML was not developed as a SPL from the beginning. Therefore, the extracted feature locations which might have an influence on feature-to-code mapping results might be inaccurate. However, as discussed above, ECCO has been evaluated for different types of artifacts. Furthermore, the quality of the pull operation is independent of the plugins used by ECCO to support different languages.

In terms of performance, one might argue that ArgoUML is not comparable with an industry-size case study. However, ArgoUML is a complex system (120 KLoC) and the performance results show that the feature-oriented operations can be integrated in the daily workflows of developers. Additionally, we successfully applied the approach to large industrial systems as reported in (Hinterreiter et al., 2020b).

7.5 Discussion

Correctness. As demonstrated in this section, we achieved very high values (close to 1.0) for both precision and recall for the feature-oriented pull operation. The minor problems we discovered were caused by the mappings automatically extracted by the variation control system, which depend on the used diffing and matching algorithms as well as the reader for the specific artifact types. An example of a program structure causing problems is a highly fragmented if-elseif cascade mapping to many different features. Due to its structure, it can be problematic to maintain correct mappings during evolution. One way to address the problem is replacing the if-elseif cascade with a switch-statement providing a clearer and more uniform structure. Another approach to eliminate this uncertainty is an improved matching algorithm. However, this problem is not subject to the distributed operation but caused by ECCO’s commit operation and the reader used for the specific artifact type (in this case Java code) and thus out of the scope of this paper.

Usefulness. Demonstrating usefulness is usually harder than proving correctness. However, in this case, we argue that the usefulness of feature-oriented distributed development operations is already evident in everyday practice. The success of distributed version control systems such as Git and platforms such as GitHub, which were adopted by open source
projects as well as large corporations, is evidence of the demand for support of distributed development. The fact that many popular branching models for Git use feature branches shows that developers try to find ways to introduce feature-oriented development into their workflows even with the lack of dedicated tool support (which this work provides). Furthermore, a recent Dagstuhl report emphasizes the need to enhance version control systems with support for variability and evolution in space (Berger et al., 2019), which is the emphasis of our work. Another piece of evidence for the usefulness and applicability of distributed feature-oriented development is related research on extracting features from forks (Zhou et al., 2018). Such work shows that current development practices already map well to feature-oriented development paradigms, despite a lack of tool support. Using feature-oriented distributed operations in the first place would make the retroactive extraction of features from forks (resulting from conventional distributed operations) obsolete.

8 Related work

Clone-and-own reuse. Several approaches provide support for creating and managing clones in product line engineering. Rubin et al. (2013) present an operator framework covering atomic operations used to manage cloned variants. For instance, they identify operations considering dependencies between features (dependsOn?), distinct implementations of similar features (same?), and conflicting features (interact?). The VariantSync project (Pfofe et al., 2016; Kehrer et al., 2021) intends to keep clones separate instead of consolidating them into a product line. It aims to support the synchronization of clones based on features. Fischer et al. (2014) originally present their ECCO (Extraction and Composition for Clone-and-Own) approach as a method to refactor cloned variants into software product lines, i.e., comparing different product variants in retrospect to extract feature-to-code traces, interactions between features, and dependencies between traces (thereby implementing some of Rubin et al.’s operations). Rabiser et al. present a modeling approach based on prototypes, i.e., prefabricated objects from which clones are created (Rabiser et al., 2016). Similar to the operations presented in this paper, the approach considers clones at different levels (products, components, features). However, the approach focuses on the variability modeling aspects and does not address aspects of distributed version control.

Variation control systems. The branching and forking mechanisms of version control systems are widely used in industrial practice to manage products, features and variabilities. Montalvillo & Díaz (2015) introduced a branching model and operations for GitHub, trying to provide better support of a version control systems to be used in SPL development. The variation control system ECCO (Linsbauer et al., 2013, 2014, 2016, 2017b) is a key component of the FORCE² platform. A similar approach is SuperMod (Schwägerl et al., 2015), which provides feature-oriented support in the area of model-driven software product line engineering. SuperMod also considers collaborative development and support for merging and solving conflicts (Schwägerl et al., 2015). Similar to our approach, SuperMod’s pull operation allows to pull evolved product lines from a remote repository. However, the distributed operations of SuperMod are not feature-aware, i.e., one cannot limit the features transferred and thus always transfers the entire remote repository.

Collaboration and awareness. Researchers have recognized awareness as an essential success factor of collaborative software development. For example, the investigation of Duc et al. (2014) on multi-platform development practices showed that diverged code
bases frequently lead to redundant development. Our approach can be used to develop an understanding of changes at the level of features in distributed development. This can contribute to overcoming the difficulties of pull-based development processes shown in empirical studies: for instance, Gousios et al. (2014) investigate reasons for not merging pull requests and show that almost one-third of unmerged pull requests are closed as no longer relevant, e.g., if a feature is already implemented in another branch or if the pull request would duplicate already existing functionality.

Visualization techniques have also been used widely and effectively to increase awareness (Dourish & Bellotti, 1992; Lettner & Grünbacher, 2015) about software evolution (Novais et al., 2013). For instance, Montalvillo et al. (2018) presented ‘peering bars’, which extend version control systems to visualize how a product’s features have been upgraded in other branches to support the merge process. Similarly, Lettner and Grünbacher (2015) proposed a publish–subscribe approach to feature-evolution tracking in software ecosystems based on feature feeds and awareness models. IBM’s Jazz software development environment provides feeds and dashboards aggregating data to improve awareness in development teams (Frost, 2007). Feeds keep developers updated on events such as build results, task modifications or approvals. Further, Holl et al. (2012) support multiple users in performing distributed product derivation of a multi-product line by sharing configuration information. While Holl et al. aim at ensuring awareness regarding the configuration choices of users configuring related PLs, our approach supports developers communicating about evolving features and artifacts and is not limited to product configuration.

Feature interactions. Interactions between software features have been investigated in multiple communities. For instance, Zave (1993) reported on the problem of feature interactions in continuously evolving systems. Dependencies and interactions between features can be inferred from the hierarchical feature models representing commonalities and variability of a system, but they also exist as cross-tree relations in the feature tree. Ferber et al. (2002) have shown that such dependencies are often difficult to represent in feature models. This gap has been addressed by Feichtinger et al. (2019), who presented an approach supporting engineers in identifying and resolving inconsistencies between features and the code implementing them. Their approach combines feature-to-code mappings, static code analysis, and a variation control system to lift complex code-level dependencies to feature models.

9 Conclusions and future work

In this paper, we presented an approach supporting distributed development via feature-oriented operations for clone and pull. While current version control systems support cloning of entire repositories, the distribution operations presented in this paper support handling variants at the level of features. Our approach is implemented in the FORCE² tool environment and relies on the variation control system ECCO. It expands our approach toward a distributed platform for managing development in multiple distributed product lines, which is highly important in software ecosystems. Our evaluation demonstrated very high precision and recall of the feature-oriented pull operation for transferring features and corresponding artifacts between different platforms. We also looked at specific pull cases to confirm the behavior for cases of interacting features. Furthermore, our performance evaluation demonstrated that the operations are fast enough to be integrated in the development workflows of engineers. By evaluating the size and scattering of features, we showed that the approach also works when mapped artifacts are
big and scattered over many locations in the source code. In particular, feature-oriented pull operations are particularly useful for features with high scattering, as manual integration of such features can represent a significant effort for developers.

As part of our future work, we plan to integrate the static code analysis of FORCE² with the clone and pull operations. Analyzing code-level feature dependencies will allow us to automatically suggest possible missing features to developers performing a clone or pull.

Acknowledgements We acknowledge the financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and Keba AG, Austria.

Funding Open access funding provided by Johannes Kepler University Linz.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article’s Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

Angerer, F., Grimmer, A., Prähofer, H., et al. (2019). Change impact analysis for maintenance and evolution of variable software systems. Automated Software Engineering, 26, 417–461.

Angerer, F., Prähofer, H., Lettner, D., et al. (2014). Identifying inactive code in product lines with configuration-aware system dependence graphs. In: Proceedings 18th Int’l Software Product Line Conference, pp 52–61.

Apel, S., Batory, D., Kästner, C., et al. (2013). Feature-Oriented Software Product Lines: Concepts and Implementation. Springer.

Berger, T., Chechik, M., Kehrer, T., et al. (2019). Software Evolution in Time and Space: Unifying Version and Variability Management (Dagstuhl Seminar 19191). Dagstuhl Reports, 9(5), 1–30.

Conradi, R., & Westfechtel, B. (1998). Version models for software configuration management. ACM Computing Surveys, 30(2), 232–282. https://doi.org/10.1145/280277.280280.

Dourish, P., & Bellotti, V. (1992). Awareness and coordination in shared workspaces. In: Proceedings ACM Conference on Computer-supported cooperative work, pp 107–114.

Duc, A. N., Mockus, A., Hackbarth, R. L., et al. (2014). Forking and coordination in multi-platform development: a case study. In: Proceedings ACM-IEEE Int’l Symposium on Empirical Software Engineering and Measurement, ESEM’14, Torino, Italy, pp 59:1–59:10.

Feichtinger, K., Hinterreiter, D., Linsbauer, L., et al. (2019). Supporting feature model evolution by suggesting constraints from code-level dependency analyses. In: Proceedings 18th Int’l Conference on Generative Programming: Concepts & Experiences, pp 129–142.

Ferber, S., Haag, J., & Savolainen, J. (2002). Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line. Software Product Lines (pp. 235–256). Berlin Heidelberg, Lecture Notes in Computer Science: Springer.

Fischer, S., Linsbauer, L., Lopez-Herrejon, R. E., et al. (2014). Enhancing clone-and-own with systematic reuse for developing software variants. In: Proceedings 30th Int’l Conference on Software Maintenance and Evolution, pp 391–400.

Frost, R. (2007). Jazz and the eclipse way of collaboration. IEEE Software, 24(6), 114–117.

Gousios, G., Pinzger, M., & van Deursen, A. (2014). An exploratory study of the pull-based software development model. In: 36th Int’l Conference on Software Engineering (ICSE), pp 345–355.

Grimmer, A., Angerer, F., Prähofer, H., et al. (2016). Supporting Program Analysis for Non-Mainstream Languages: Experiences and Lessons Learned. In: Proceedings 23rd SANER Conference, pp 460–469.
Grünbacher, P., Hanl, R., & Linsbauer, L. (2021). Using music features for managing revisions and variants in music notation software. In: Gottfried R, Hajdu G, Sello I, et al (eds) Proceedings of the International Conference on Technologies for Music Notation and Representation – TENOR’20/21. Hamburg University for Music and Theater, Hamburg, Germany, pp 212–220.

Hinterreiter, D., Grünbacher, P., & Práhofer, H. (2020a). Visualizing feature-level evolution in product lines: A research preview. In: Madhavji N, Pasquale L (eds) Proceedings of the 26th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ), Essen, Germany, pp 1–7. https://doi.org/10.1007/978-3-030-44429-7_21

Hinterreiter, D., Linsbauer, L., Feichtinger, K., et al. (2020b). Supporting feature-oriented evolution in industrial automation product lines. *Concurrent Engineering: Research and Applications*, 28:265–279.

Hinterreiter, D., Linsbauer, L., Grünbacher, P., et al. (2021). Feature-oriented clone and pull for distributed development and evolution. In: Proceedings of the 14th Int’l Conference on the Quality of Information and Communications Technology (QUATIC), Online, pp 67–81. https://doi.org/10.1007/978-3-030-85347-1_6

Hinterreiter, D., Linsbauer, L., Reisinger, F., et al. (2018). Feature-oriented evolution of automation software systems in industrial software ecosystems. In: Proceedings 23rd IEEE Int’l Conference on Emerging Technologies and Factory Automation, pp 107–114.

Hinterreiter, D., Nieke, M., Linsbauer, L., et al. (2019). Harmonized temporal feature modeling to uniformly perform, track, analyze, and replay software product line evolution. In: Proceedings of the 18th ACM SIGPLAN Int’l Conference on Generative Programming: Concepts and Experiences, New York, NY, USA, GPCE 2019, p 115–128.

Holl, G., Grünbacher, P., Elsner, C., et al. (2012). Supporting awareness during collaborative and distributed configuration of multi product lines. In: K. R. P. H. Leung & P. Muenchaisri (Eds.), *Proceedings of the 19th IEEE Asia-Pacific Software Engineering Conference* (pp. 137–147). APSEC 2012, Hong Kong, China, December 4–7.

Kehrer, T., Thüm, T., Schultheiß, A., et al. (2021). Bridging the gap between clone-and-own and software product lines. In: ICSE (NIER). IEEE, pp 21–25.

Lettner, D., & Grünbacher, P. (2015). Using feature feeds to improve developer awareness in software ecosystem evolution. *Proceedings 9th Int’l Workshop on Variability Modelling of Software-intensive Systems* (pp. 11–18). Hildesheim, Germany: ACM.

Linsbauer, L., Angerer, F., Grünbacher, P., et al. (2014). Recovering feature-to-code mappings in mixed-variability software systems. In: Proceedings 30th Int’l Conference on Software Maintenance and Evolution, pp 426–430.

Linsbauer, L., Berger, T., & Grünbacher, P. (2017a). A classification of variation control systems. In: Proceedings of the 16th Int’l Conference on Generative Programming: Concepts & Experience (GPCE’17). ACM, New York, NY, USA, pp 49–62.

Linsbauer, L., Egyed, A., & Lopez-Herrejon, R. E. (2016). A Variability-Aware Configuration Management and Revision Control Platform. In: Proceedings 38th Int’l Conf. on Software Engineering (Companion), pp 803–806.

Linsbauer, L., Fischer, S., Michelon, G. K., et al. (2022). Systematic software reuse with automated extraction and composition for clone-and-own. In: R. E. Lopez-Herrejon, J. Martinez, T. Ziad, et al. (Eds.), *Handbook of Re-Engineering Software Intensive Systems into Software Product Lines*. Springer.

Linsbauer, L., Lopez-Herrejon, R. E., & Egyed, A. (2017b). Variability extraction and modeling for product variants. *Software and System Modeling, 16*(4):1179–1199.

Linsbauer, L., Lopez-Herrejon, R. E., & Egyed, A. (2013). Recovering traceability between features and code in product variants. *Proceedings 17th Software Product Line Conference* (pp. 131–140). Japan: Tokyo.

Linsbauer, L., Schwägerl, F., Berger, T., et al. (2021). Concepts of variation control systems. *Journal of Systems and Software, 171*(110), 796.

Martinez, J., Ordoñez, N., Ternava, X., et al. (2018). Feature location benchmark with ArgoUML SPL. In: Proceedings 22nd Int’l Systems and Software Product Line Conference, pp 257–263.

Michelon, G. K., Linsbauer, L., Assunção, W. K. G., et al. (2019). Comparison-based feature location in ArgoUML variants. In: Proceedings 23rd Int’l Systems and Software Product Line Conference, pp 93–97.

Montalvillo, L., & Diaz, O. (2015). Tuning GitHub for SPL development: Branching models & repository operations for product engineers. In: Proceedings of the 19th Int’l Conference on Software Product Line. ACM, New York, NY, USA, SPLC ’15, pp 111–120.

Montalvillo, L., Diaz, O., & Fogdal, T. (2018). Reducing coordination overhead in SPLs: peering in on peers. In: Proceedings of the 22nd Int’l Systems and Software Product Line Conference, pp 110–120.
Novais, R. L., Torres, A., Mendes, T. S., et al. (2013). Software evolution visualization: A systematic mapping study. *Information & Software Technology, 55*(11), 1860–1883.

OMG. (2011). Abstract syntax tree metamodel (astm) version 1.0. URL https://www.omg.org/spec/ASTM/1.0/PDF

Pfofe, T., Thüm, T., Schulze, S., et al. (2016). Synchronizing software variants with variantSync. In: SPLC. ACM, pp 329–332.

Pohl, K., Böckle, G., & van der Linden, F. J. (2005). *Software Product Line Engineering: Foundations, Principles, and Techniques*. Berlin: Springer.

Rabiser, D., Grünbacher, P., Prähofer, H., et al. (2016). A prototype-based approach for managing clones in clone-and-own product lines. *Proceedings 20th Int’l Software Product Line Conference* (pp. 35–44). China: Beijing.

Ramirez, A., Vanpeperstraete, P., Rueckert, A., et al. (2011). ArgoUML user manual. Technical Report.

Rubin, J., Czarnecki, K., Chechik, M. (2013). Managing cloned variants: a framework and experience. In: Proceedings 17th Int’l Software Product Line Conference, pp 101–110.

Schwägerl, F., Buchmann, T., Uhrig, S., et al. (2015). Towards the integration of model-driven engineering, software product line engineering, and software configuration management. In: 3rd Int’l Conference on Model-Driven Engineering and Software Development (MODELSWARD), pp 1–14.

Schwägerl, F., Buchmann, T., & Westfechtel, B. (2015). Supermod – A model-driven tool that combines version control and software product line engineering. In: Proceedings of the 10th Int’l Conference on Software Paradigm Trends, pp 5–18.

Schwägerl, F., & Westfechtel, B. (2019). Integrated revision and variation control for evolving model-driven software product lines. *Software & Systems Modeling, 18*(6), 3373–3420.

Zave, P. (1993). Feature interactions and formal specifications in telecommunications. *Computer, 26*(8), 20–28.

Zhou, S., Stanciulescu, S., Leßenich, O., et al. (2018). Identifying features in forks. In: Proceedings 40th Int’l Conference on Software Engineering, pp 105–116.

Zhou, S., Vasilescu, B., & Kästner, C. (2019). What the fork: A study of inefficient and efficient forking practices in social coding. In: Proceedings 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, p 350–361.

**Publisher’s Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Daniel Hinterreiter was a researcher in the Christian Doppler Laboratory for Monitoring and Evolution of Very-Large-Scale Software Systems at Johannes Kepler University Linz, Austria (JKU). He received a master’s degree and doctoral degree from JKU. His current research interests include feature modeling and software evolution in industrial software ecosystems.
Lukas Linsbauer is a postdoctoral researcher at the Technical University of Braunschweig in Germany. His research interests include software product lines, variability modeling and management, feature-oriented software and systems development, highly variable and configurable systems, traceability, and version control systems. His most recent research is in the area of variation control systems. He received his Ph.D. in Computer Science (Software Engineering) in 2016 from the Johannes Kepler University Linz, Austria.

Herbert Prähofer is an Associate Professor at the Institute for System Software at the JKU. His research interests include software development methods and tools, model-based software development, simulation theory and methods, and software engineering methods in the automation domain. He received a master’s degree and a doctoral degree in computer science from JKU in 1986 and 1991, respectively.

Paul Grünbacher is an Associate Professor at the Institute for Software Systems Engineering at Johannes Kepler Universität Linz (Austria). His research interests include software product lines, model-based development and evolution, requirements engineering, and software monitoring. Paul is an Editorial Board Member of the Empirical Software Engineering Journal (Springer) and the Information and Software Technology Journal (Elsevier).