Author preprint for a paper published in the proceedings of the 43rd International Conference on Software Engineering (ICSE 2021)

Seamless Variability Management With the Virtual Platform

Wardah Mahmood*, Daniel Strüber†, Thorsten Berger*‡, Ralf Lämmel§, and Mukelabai Mukelabai*
*Chalmers | University of Gothenburg, Sweden
†Radboud University, Netherlands
‡Ruhr University Bochum, Germany
§University of Koblenz-Landau, Germany

***Abstract*—Customization is a general trend in software engineering, demanding systems that support variable stakeholder requirements. Two opposing strategies are commonly used to create variants: software clone&own and software configuration with an integrated platform. Organizations often start with the former, which is cheap, agile, and supports quick innovation, but does not scale. The latter scales by establishing an integrated platform that shares software assets between variants, but requires high up-front investments or risky migration processes. So, could we have a method that allows an easy transition or even combine the benefits of both strategies? We propose a method and tool that supports a truly incremental development of variant-rich systems, exploiting a spectrum between both opposing strategies. We design, formalize, and prototype the variability-management framework *virtual platform*. It bridges clone&own and platform-oriented development. Relying on programming-language-independent conceptual structures representing software assets, it offers operators for engineering and evolving a system, comprising: traditional, asset-oriented operators and novel, feature-oriented operators for incrementally adopting concepts of an integrated platform. The operators record meta-data that is exploited by other operators to support the transition. Among others, they eliminate expensive feature-location effort or the need to trace clones. Our evaluation simulates the evolution of a real-world, clone-based system, measuring its costs and benefits.**

***Index Terms*—software product lines, variability management, clone management, re-engineering, framework**

I. Introduction

Software systems often need to exist in many different variants. Organizations create variants to adapt systems to varying stakeholder requirements—for instance, to address a variety of market segments, runtime environments or different hardware. Creating variants allows organizations to experiment with new ideas and to test them on the market, which easily leads to a portfolio of system variants that needs to be maintained.

Two opposing strategies exist for engineering variants. A convenient and frequent strategy is *clone&own* [1], [2], [3], [4], [5], where developers create one system and then clone and adapt it to the new requirements. This strategy is well-supported by current version-control systems and tools, such as GIT, relying on their forking, branching, merging, and pull request facilities. The frequent adoption of clone&own [6], [1], [4] is usually attributed to its inexpensiveness, flexibility, and provided developer independence. However, clone&own does not scale with the number of variants and then imposes substantial maintenance overheads. A scalable strategy is to integrate the cloned variants into a *configurable and integrated platform*, by adopting platform-oriented engineering methods, such as software product line engineering (SPLE) [7], [8], [9], [10], [11]. Individual variants are then derived by configuring the platform. This strategy is typically advocated for systems with many variants, such as software product lines (e.g., automotive/avionics control systems and industrial automation systems) or highly configurable systems (e.g., the Linux kernel). This strategy scales, but is often difficult to adopt and requires substantial up-front investments, since variability concepts (e.g., a feature model [12], [13], feature-to-asset traceability [14], [15], a configuration tool [16]) need to be introduced and assets made reusable or configurable. In practice, organizations often start with clone&own and later face the need to migrate to a platform in a risky and costly process [17], [18], [6], [19], recovering meta-data that was never recorded during clone&own, such as features and their locations in software assets [20], [13].

Over the last decades, researchers focused on heuristic techniques to recover information from legacy codebases, including feature identification [21], [22], [23], feature location [24], [25], [26], variability mining [27], [28], and clone-detection techniques [29], [30]. Unfortunately, such techniques are usually not accurate enough to be applicable in practice, and also require substantial effort to set them up and provide with manual input (e.g., specific program entry points for feature location techniques [31]). As we will show, existing platform migration techniques either heavily rely on such heuristics or have only been formulated as abstract frameworks so far. Moreover, they tend to prescribe non-iterative, waterfall-like migrations, making it risky and expensive.

We take a different route and present a method to continuously record the relevant meta-data already during clone&own, and to incrementally transition towards a more scalable platform-oriented strategy, exploiting the meta-data recorded. We design, formalize, and prototype a lightweight method called *virtual platform*, generalizing clone-management and product-line migration frameworks. We exploit a spectrum between the two extremes of ad hoc clone&own and fully integrated platform, supporting both kinds of development. As such, the virtual platform bridges clone&own and platform-oriented development (SPLE). Based on the number of variants, organizations can decide to use only a subset of all the variability-implementation concepts that are typically

required for an integrated platform. This allows organizations to be flexible and innovative by starting with clone&own and then incrementally adopting the variability-implementation concepts necessary to scale the development, as indicated by industrial practices for product-line adoption [32], [11], [33], [34]. This realizes an incremental adoption of platforms with incremental benefits for incremental investment. Furthermore, it also allows to use clone&own even when a platform is already established, to support a more agile development with cloning and quickly prototyping new variants. The framework is lightweight, since it avoids upfront investments and can be easily integrated with version-control systems or IDEs, where its operators can be mapped to existing activities, avoiding extra effort. This way, our new (feature-oriented) operators are cheap to invoke during development, when the feature knowledge is still fresh in the developer's mind, allowing to record meta-data in a lightweight way.

The term "virtual platform" was introduced earlier in a short paper [35] discussing an incremental migration of clone-based variants into a platform. It introduced governance levels reflecting a spectrum between the two extremes ad hoc clone&own and fully integrated platform. Higher levels involve a super-set of the variability concepts of lower levels. Advancing a level—e.g., when the number of variants increases—supports an incremental adoption of variability concepts, avoiding the costly and risky "big bang" migration [17] often leading to re-engineering efforts over years [18], [36]. This early, high-level description of a strategy to incrementally scale the management of variants paved the way for this paper.

One of our core contributions are conceptual structures and formalized operators for the virtual platform, which are related to ordinary code editing, but also record and exploit meta-data. While we prototypically implemented the virtual platform on top of an ordinary file system, our work gives rise to realize it upon a database (to enhance scalability), within an integrated development environment (IDE), or as a command-line tool. The meta-data could also easily be saved directly in the software-assets using lightweight embedded annotations (as our prototype does).

We evaluated our prototype on a reasonably sized system (57.4k lines of text, 4 variants), where we simulated evolution activities that are typical of practical software systems. Our prototype was able to fully simulate and manage all considered activities. From a cost-benefit analysis, we conclude that the virtual platform offers significant cost savings during inevitable evolution and maintenance activities.

In summary, we contribute:

- **a mechanization** of the so-far abstract idea of operators mediating between clone&own and an integrated platform, defined upon conceptual, language-independent structures,
- **a prototype of the virtual platform [37]** in Scala,
- **a comparative evaluation of the virtual platform** against five related frameworks, based on their ability to support common evolution scenarios,
- **a cost-and-benefit evaluation of the virtual platform**, based on a simulation study featuring the revision history of a real variant-rich open-source system, and
- **an online appendix [38]** with a technical report about our operators, additional examples, and evaluation data.

II. Motivation and Overview

We provide a core scenario of seamless variability management as a running example and an overview of the virtual platform. While rooted in a deliberately simple application domain, the example is inspired by documented real product-line migration scenarios [19], [39]. It includes tasks that are tedious and error-prone in practice (e.g., bugfix propagation along branches).

A. Motivating Running Example

We now discuss relevant problems of managing variants inspired by actual industrial practices, also presenting our solution in the virtual platform and how a developer would use the virtual platform. Specifically, developers interact with the virtual platform by invoking its provided operators, either via the command-line or an integration with an IDE or version-control system provided by a tool vendor (see Sec. II-B for details). While the traditional, asset-oriented operators can run transparently in the background, only the feature-oriented operators require an extra user interaction for invoking the operators. The operator are described in detail in Sec. V.

Consider the scenario of an organization developing and evolving variants of a calculator tool. Our organization starts creating a project of a simple calculator called *BasicCalculator* (BC) that supports basic arithmetics: *addition*, *subtraction*, *multiplication*, and *division*. Soon, based on customer requests, the organization needs to create variants of *BC*, which have substantial commonalities, but also differ in functional aspects.

Figure 1 illustrates the two opposing strategies (cf. Sec. I) for realizing the variants. Specifically, it shows two alternate realizations of a variant of *BasicCalculator* with a small display, requiring the rounding of results (feature *SmallDisplay*). To the left, the code is cloned and adapted (one line changed in the branch BC+SmallDisplay); to the right, a configuration option represents the change in a common codebase (integrated platform). The changes are usually more complex (e.g., features can be highly scattered [40], [41]), as well as the representation of variability in the integrated platform. We also need more variability concepts, among others, features [42], [43], [44], code-level configuration [10], feature-to-asset traceability [14], [15], [45], a feature model (a hierarchical structure with features and their dependencies) [12], [13], a configurable build system [10], and a configurator tool [16], [46], [47]. This example shows that, when it becomes necessary to migrate from clone&own to an integrated platform, important information needs to be recovered, specifically: that a feature *SmallDisplay* was implemented and where its code is located. Recovering such information in systems with many features and sizable codebases is laborious, time-consuming, and inaccurate at best. Also, migration can be invasive, risky, and costly, especially hard to achieve under market pressure [48], [18], [36], [20], [17], [49].

The virtual platform exploits a spectrum between the two extremes and supports an incremental transition as shown in

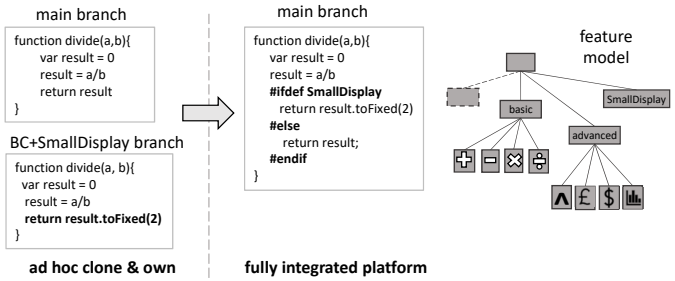

Fig. 1. Ad hoc clone&own vs. fully integrated platform illustrated for two variants: the *BasicCalculator* and a variant with only a small display

Fig. 2. It adapts the governance levels from prior work [35], which also explains the benefits of each transition step in detail.

Let us further discuss the evolution of our calculator using **ad hoc clone&own**. After the *BasicCalculator* and a variant of it for small displays (BC+SmallDisplay) is created, customers request a *ScientificCalculator*, which should solve complex inputs, such as *expressions*, *factorials*, and *logarithms*. Our organization decides to copy and adapt the codebase from *BasicCalculator*, since there is no need for a *ScientificCalculator* with small display support; otherwise we would already have four cloned variants. As such, cloning provides a baseline minimizing the duplication of efforts. Soon after, the organization needs to create another variant called *GraphingCalculator*, for which it selects the most similar variant, *ScientificCalculator*, and clones and adapts it. It also notices that some functionality in *BasicCalculator* had in the meantime received a bug fix, which the organization also applies to *GraphingCalculator*, now realizing that also *ScientificCalculator* needs to receive the bug fix.

Problem 1: Where are my clones? With many more variants developed using ad hoc clone&own, developers lose overview. If a change (e.g., a bug fix) is to be replicated, developers need to recover which project was cloned from which, in the worst case requiring a clone-detection technique. Also, the added effort in synchronizing cloned implementations is likely to surpass the initial benefit of reuse via cloning.

Solution 1: Clone&own with provenance (Fig. 2, 1st level). Our solution is to record traceability information about the cloned variants' provenance, which eases tracking and synchronizing clones. It also bypasses the inaccuracies associated with clone detection, making tasks such as change propagation more effective. The virtual platform records clone traces among assets in the background, without requiring extra effort from the developer, but who can query it for obtaining the clones of an asset.

To this end, the developer invokes the `CloneAsset` operator provided by the virtual platform. As a result, a trace between the original asset and its clone is stored in a trace database, which can be queried at any time by the developer to retrieve clones of an asset quickly and accurately. The developer can later propagate changes between the original asset and its clone (`PropagateToAsset`) or integrate changes between the assets (either manually or using a tool) by exploiting the continuously recorded meta-data.

Problem 2: What is in my cloned variants? With more variants, despite provenance information, the problem arises that developers lose overview. To understand what is in the variants, we need a more abstract representation of assets. For cloning, this is also necessary to select an existing variant closest to the desired one in terms of the desired features. Furthermore, our organization finds the feature *exponent* developed in *ScientificCalculator* to be useful for other cloned variants. To clone it, the developer needs to know which implementation assets belong to the feature.

Solution 2: Clone&own with features (Fig. 2, 2nd level). Adding feature meta-data adds perspective and allows functional decomposition. It also allows representing assets in terms of features, to reuse and clone features across projects. Lastly, including feature-related information allows going past the efforts and inaccuracies of *feature location* (recovering where a feature is implemented), making feature reuse and maintenance more effective. The virtual platform offers operators to add features conveniently (at the same time annotating assets).

The developer maps assets to features by using the operator `MapAssetToFeature`. She can later query the virtual platform to find the location of the features using the operator `getMappedAssets`, and also to clone assets along with feature mappings (`CloneAsset`).

Problem 3: How to reduce redundancy? Despite features, which help maintaining variants, substantial redundancy exists.

Solution 3: Clone&own with configuration (Fig. 2, 3rd level). To reduce it, our organization starts to incorporate configuration mechanisms. These allow to enable or disable features, such as *SmallDisplay*, which control variation points. This reduces redundancy and maximizes reuse. So, the organization maintains a list of features and uses a configurator tool. The virtual platform supports this solution with a simple operator.

Over time, the developer adds features by invoking the operator `AddFeature`. She can map the assets to features using `MapAssetToFeature` and clone features using `CloneFeature`. She can also make features optional by invoking `MakeFeatureOptional`. Variants can be configured by cloning the repository (`CloneAsset`) with assets mapped to only the selected features (`getMappedAssets`).

Problem 4: How to keep an overview over the features? The more features and variation points the organization incorporates, the more it loses overview over the features and their relationships, including feature dependencies (accidentally ignoring those can lead to invalid variants). Maintaining such information would also help scoping variants.

Solution 4: Clone&own with a feature model (Fig. 2, 4th level). Our organization introduces a feature model, which captures features and their constraints, also as input to the configurator. Feature models are very intuitive and simple models, which provide deep insights without much additional tool support. They also foster communication among stakeholders and validate feature configurations. With this solution, consistency between features and clones is high, since developers can also exploit the clone traces and use the virtual platform for feature-based change propagation.

The developer adds a feature model to the repository with the operator `AddFeatureModelToAsset`. She can change the feature model to add and remove features at any

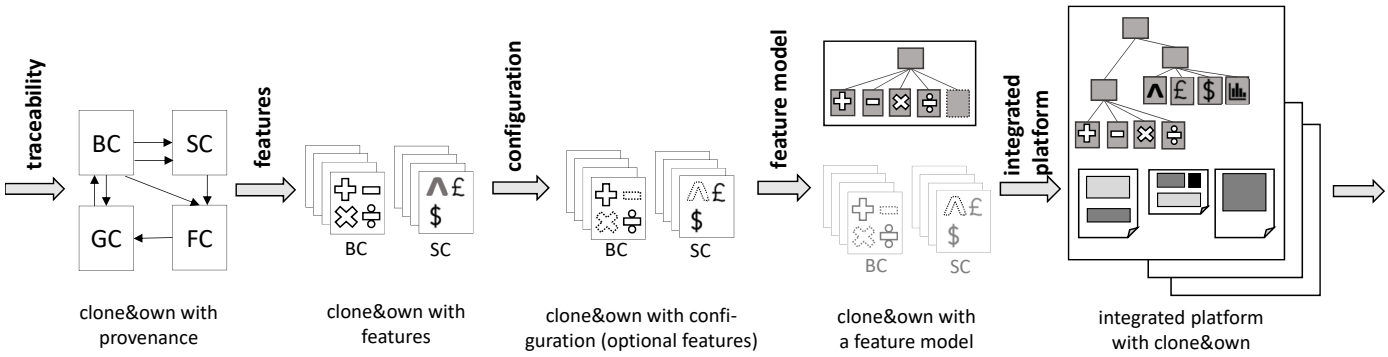

Fig. 2. Spectrum between the extremes ad hoc clone&own and a fully integrated platform (see Fig. 1 for both), illustrated with cloned variants: *BasicCalculator* (BC), *ScientificCalculator* (SC), *GraphingCalculator* (GC), and *FinancialCalculator* (FC). The virtual platform provides operators to transition along this spectrum (e.g., to incrementally adopt a platform).

time. She can map assets to features from the feature model (`MapAssetToFeature`), clone features across projects (`CloneFeature`), and propagate changes in features to their clones (`PropagateToFeature`).

Problem 5: How to keep consistency, improve quality, and further reduce redundancy? Our organization needs to further scale the development with an ever-increasing number of variants (due to rapidly changing market needs), while it has problems maintaining consistency and propagating changes, despite some redundancy already being reduced with Solution 3. It is also likely that eventually, there will be some projects with a configuration mechanism and some without.

Solution 5: Integrated platform with clone&own (Fig. 2, 5th level). Our organization integrates the projects into a consolidated platform. Luckily it can exploit meta-data about clone traceability (*provenance*) and features with their locations in assets. The virtual platform provides support for this kind of information, easing the integration of cloned variants into a platform. Of course, developers might have forgotten to record all that information, then it is natural to recover it. As long as some information is recorded, a benefit arises in terms of saved feature identification, feature location and clone-detection effort.

B. Virtual Platform Overview

Our goal is to combine the benefits of the two opposing strategies clone&own and integrated platform, exploiting a spectrum between both and allowing incremental transition as in our running example (Sec. II-A). To this end, we designed a framework called virtual platform comprising conceptual structures upon which operators modifying the structures are executed by developers. The conceptual structures abstractly represent software assets at various levels of granularity—from whole repositories to blocks of code—and can be adapted to specific asset languages (explained shortly in Sec. IV).

In addition, they maintain information about variability, specifically feature information, feature-to-asset mappings, and clone traces. The virtual platform extends other development tools, specifically, IDEs and version control systems. On top of these, which are concerned with the management of *assets*, the virtual platform provides dedicated functionality for managing *features*. Operators can be either traditional, meaning they are concerned with asset management, or feature-oriented, meaning they are devoted to features and their locations in assets. In contrast to traditional development workflows, the use of dedicated feature-oriented operators incurs a certain cost, but promises benefits to developers. In Sec. VI, we study this trade-off.

Figure 3 illustrates interactions and internal workings of the virtual platform. Developers can interact with it directly or indirectly. The former is enabled via extensions and hooks of existing tools. Specifically, traditional IDE commands such as *"Create File"* and version-control commands such as *"Add File"* are linked to the traditional, asset-oriented operators of the virtual platform (e.g., *"Create Asset"*) and do not impose additional effort for developers. Feature-oriented operators can be implemented by new, feature-oriented IDE commands (e.g., *"Create Feature"*). Direct interaction is enabled via a command-line interface, where developers can call feature-oriented operations such as *"Create Feature"* directly.

III. Methodology

We followed a design-science-like strategy to iteratively define the conceptual structures, the operators, and to evaluate them using unit tests representing common scenarios. Specifically, for the structures and operators, we aimed at maximizing the support for different scenarios from the literature and our own professional experience. The main challenge was to define adequate structures that, while programming-language-independent, can be mapped to many of the different asset types of real-world software projects, as well as to design the operators to be able to operate on the structures.

Initial Design. We started by analyzing clone-management and platform-migration frameworks proposed in the literature, from which we extracted development activities that should be supported by the virtual platform. We also had a series of

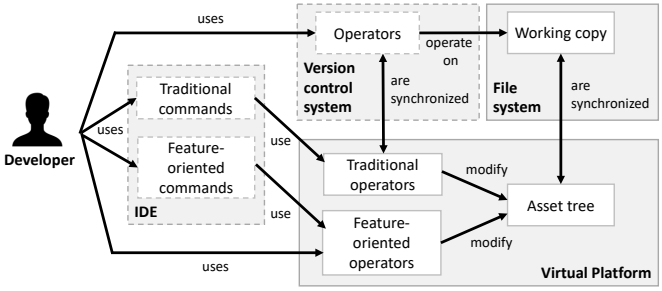

Fig. 3. Overview (dashed boxes represent optional parts)

discussions among the authors, one from industry and four from academia. Two authors have over ten years of research experience in variability management and SPLE. We also created ad hoc examples in the discussion meetings. From these sources, we identified an initial set of data structures and operators, and implemented them in Scala.

Specifically, from the literature, we identified five relevant works on clone management and product-line migration using our expert knowledge. Rubin et al.'s product-line migration framework [50], [51] offers operators that support the narrative that a mechanization—i.e., an operator-based perspective—leads to more efficient implementation and support. Fischer et al.'s [52] framework and tool ECCO relies on heuristics to identify commonalities and allows composing new product variants using reusable assets. Martinez et al.'s tool BUT4Reuse [53] is an extraction-based technique for product-line migration, including support for feature-model synthesis. Pfofe et al.'s tool VariantSync [54] supports clone-management by easing the synchronization of assets among cloned variants. Montalvillo et al.'s operators and branching models for clone management in version-control systems [55] allow isolated variant development with change propagation, but without using the notion of features, as opposed to the other frameworks. For brevity, we will present the identified activities only at the end in Sec. VI-A. Detailed descriptions are in our online appendix [38].

Continuous Evaluation. Once every operator was implemented, we tested it with unit tests based on scenarios from the literature and our own experiences. We ensured that the operators assured the *well-formedness* of the conceptual structures by prohibiting illegal actions, e.g., limiting asset addition to scopes that can host an asset of the given type.

Final Qualitative and Quantitative Evaluation. We evaluated the virtual platform qualitatively by comparing it against the existing frameworks discussed above, from which we had extracted activities supported by techniques for supporting clone&own or the migration of cloned variants to an integrated platform. We evaluated the virtual platform quantitatively in a cost-benefit calculation based on simulating the development of a real open-source system developed using clone&own.

IV. Conceptual Structures

The virtual platform's conceptual structures form the basis for its operators, which we formulated as functions with side effects (in-place transformations) that modify the structures. Figure 4 illustrates the main structures and their relationships. We define them abstractly, but also provide a concrete implementation for handling assets within a file system and special support for textual files that follow a hierarchical structure (e.g., with nested classes, methods or code blocks; cf. Sec. VI).

Asset Tree (AT) is our main conceptual structure and abstractly represents a hierarchy of assets, such as the folder hierarchy, but also the hierarchy within source files. In Fig. 4, the AT is represented implicitly in the form of assets with their sub-asset relationships. The idea of AT is inspired by feature structure trees (FSTs, [56]), which represent source files. In our case, we define the AT as a hierarchical, non-cyclic tree

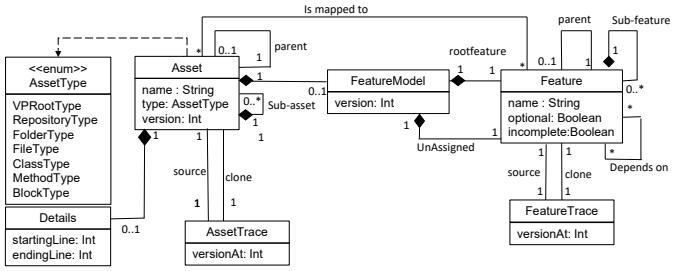

Fig. 4. Conceptual structures: asset tree, features, mappings, and clone traces

structure of nodes. It has a synthetic root node (*root*) and then represents a hierarchy that can start with repositories as the top-level nodes, followed by folders and files, and can then go into the nesting structure of elements of hierarchical files.

Every node represents an *asset* related to the project, such as a folder, a file (e.g., image, source file, model or requirements document), or text. Every *asset* has a *name*, a type (*AssetType*), and a *version* (a simple means to identify changes). An asset can have any number of *sub-assets*. It also owns a *parent* pointer p, which should define a tree, with a virtual root node (*asset* of type *VPRootType*) denoted as *root*. The *AssetType* is used to capture the role of the *asset* in the project, and can be one of the following: *VPRootType*, *RepositoryType*, *FolderType*, *FileType*, *ClassType*, *MethodType*, and *BlockType*. The type *VPRootType* is only used once in the AT, to specify the synthetic root node. The main purpose of this root node is to carry a global version (we explain versioning shortly).

Traditional SPLE architectures have a feature model per project, which can be difficult to maintain and evolve in large systems (e.g., Linux kernel [57]). We provide a more flexible structure by including an optional feature model as part of every *asset* (see composition of *feature model* in *asset* in Fig. 4).

Well-Formedness Criteria We define a partial order of valid containment over the types of assets in a check function $\mathit{containable} : \mathit{asset} \times \mathit{asset} \to \mathbb{B}$ that validates the containment based on the asset types. For instance, *VPRootType* can only be at the root, and a *MethodType* can be contained in a *FileType*, but not the other way around. Operators are implemented with consideration of *well-formedness* criteria, to ensure that the tree structure of AT is retained.

Features and Feature Models A *feature* has a *name* and two `Boolean` parameters: *optional* and *incomplete*. The field *optional* specifies whether the *feature* is mandatory or optional; *incomplete* captures information about the completeness of the *feature's* implementation. If the *feature* was cloned from another *feature model* scope, it is `true` if the new scope containing the *feature* also contains all the assets to which the *feature* is mapped; otherwise it is always `false`. Every *feature* has an optional *parent*, and any number of *sub-features*. Features can have dependencies to each other.

A *feature model* has a *root feature* and a mandatory *feature* called *Unassigned*, which contains all features that are added to the model as a result of asset cloning. That is, if any *feature* mapped to the *asset* is not present in the target *feature model* already, it is mounted under *Unassigned* (and requires developer intervention to move it to the desired location in the model).

Asset-To-Feature Mappings, in practice, can have two seman-

tics. They can be simple mapping relationships, indicating that *asset* realizes a *feature* [58]. They can also indicate variability [59], where the *asset* is included in a concrete variant if the *feature* is selected (interestingly, if an *asset* is optional based on a *feature*, then the *asset* also realizes it, but not necessarily all assets realizing a *feature* are optional). The SPLE community usually focused on the variability relationship, and the feature-location community on traceability. For the virtual platform, we unified the mechanism with which assets are mapped to features. Specifically, an *asset* has a presence condition (PC)—a propositional formula over features. A PC allows conveniently mapping assets of different granularity levels (*AssetType*) to entire *feature* expressions. Whether this relationship to the *feature* represents variability or traceability is solely determined by the *feature's optional* parameter.

Versioning of Assets. Assets (and features) have a *version*—an integer used to recognize changes in the AT (and FM), especially among cloned assets. The *version* of the *VPRootType* node has a special role, which we call "*GlobalVersion*" and which carries the most up-to-date *version*, to recognize any change in the whole AT. For simplicity, we assume that any *asset* outside the tree has a *version* of 0. After addition, it takes the *version* of the global root (initialized with 1 and incremented after any update in the AT). Versions are incremented after every modification and addition or removal of *sub-assets*. This simple versioning strategy is a sweet spot between two other alternatives: First, after every change in an *asset*, increment the *version* of the *asset* and continue updating the ancestors up to the root. This would make the tracking of the changes easy, but change propagation expensive and redundant. Second, keep two separate numbers, one global version, and one local version for every asset. This solution would ease change propagation, but yield a hard-to-understand versioning model.

Clone Traceability. To maintain trace links between source assets and their clones, we define an *AssetTraceDatabase*—essentially a list of *AssetTraces* (Fig. 4). An *AssetTrace* is a triplet of the source *asset*, its clone, and a *version* at which the source *asset* was cloned. Similarly, *feature* traces are used to keep track of the *feature* clones, and they are stored in a *FeatureTraceDatabase*. A *FeatureTrace* is also a triplet pointing to the source *feature*, its clone, and *version* at the time of cloning. These traces are a core component of our contribution, and have special relevance in cloning and change propagation for both assets and features. For brevity, we refer to both *AssetTraceDatabase* and *FeatureTraceDatabase* as *TraceDatabase* in the remainder of the paper.

V. Virtual Platform Operators

We now present the traditional, asset-oriented and the feature-oriented operators. Their underlying algorithms and further illustrations (supplementary to the illustrations used here) are provided in our online appendix [38]. The appendix also presents a number of additional *convenience operators*—utility methods that efficiently traverse the trees (AT and *feature model*) to return data that needs to be frequently accessed (such as assets mapped to a *feature* and clones of an *asset* etc).

A. Traditional/Asset-Oriented Operators

We represent conventional activities performed by developers using asset-oriented operators. These operators allow to keep the AT in sync with the working directory. Also, the assets act as *mappable* components to the features, and allow cloning and change propagation. In what follows, we introduce the asset-oriented operators with their parameter types, a brief description, and sample scenarios, inspired from our calculator running example (cf. Sec. II-A). The notation used for visualizing various scenarios is shown in Fig. 5.

`AddAsset` : *asset* $\times$ *asset* $\rightarrow \mathbb{B}$

Description: When a source *asset* (S) is added in any target *asset* (T) to a repository (e.g., a file to a folder), `AddAsset` creates an *asset* for S and adds it to the preexisting *asset* T in the AT. Additionally, it increments the *GlobalVersion*, and assigns it to S and T. This implies that the most recently changed assets are S and T. Also, it adds any *feature* mapped to S in T's *feature model* (typically repository *feature model*).

Example: Consider the *BasicCalculator (BC)* example. The developer adds the implementation for the *divide* method in the file *Operators.js*, with an annotation for the *feature DIV*. Consequently, the virtual platform creates and adds the *asset divide* (S) of *MethodType* to the *asset Operators.js* (T) of *FileType*, and *DIV* to the *feature model* of T. The *GlobalVersion* (previously 3) is incremented and assigned to *divide* and *Operators.js*. Figure 6 illustrates the scenario.

`ChangeAsset` : *asset* $\rightarrow \mathbb{B}$

Description: Upon a change in an *asset* S in the repository, `ChangeAsset` increments the *GlobalVersion* of the AT and assigns it to S. Versionable changes include renaming, addition, mapping to a *feature* and modification or removal of lines.

`RemoveAsset` : *asset* $\rightarrow \mathbb{B}$

Description: If an *asset* is deleted from a parent asset T, `RemoveAsset` removes its corresponding *asset* S in the AT, along with all its sub-assets. It increments the *GlobalVersion* and assigns it to T. Additionally, any *feature* mapped to S is also removed from the *feature model* of S if S the *only asset* mapped to it. This enforces that if all assets mapped to a *feature* are deleted, the *feature* is also deleted.

`MoveAsset` : *asset* $\times$ *asset* $\rightarrow \mathbb{B}$

Description: If an *asset* is moved from one location to another, `MoveAsset` clones the corresponding *asset* S to the new target *asset* T (using `CloneAsset`), and removes it from the sub-assets of its previous parent (using `RemoveAsset`).

Thus far, the operators we presented serve two purposes: keeping the AT synchronized with the project, and keeping

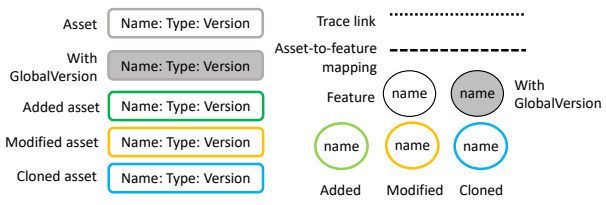

Fig. 5. Notations used in operator illustrations

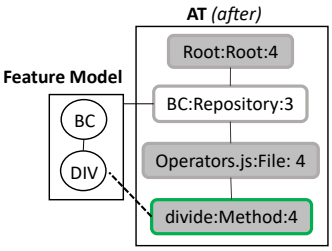

Fig. 6. Illustration of `AddAsset`(*divide*,*Operators.js*)

track of changes through versioning. Following, the operators serve two additional purposes: storing feature-oriented data, and recording traceability among clones. The exploitation of these meta-data are the essence of our framework.

`MapAssetToFeature` : *asset* $\times$ *feature* $\rightarrow \mathbb{B}$

Description: Upon addition of a *feature* mapping by a developer, `MapAssetToFeature` checks if the *feature* exists in the *feature model* of the *asset*. If not, it creates a *feature* F (with the name used by the developer), maps it to S (corresponding *asset* in the AT), and adds F to the *Unassigned feature* in the *feature model* of S. If F already exists, it simply maps F to S. For mapping, it adds F to the *presencecondition* of S with a logical disjunction. To track this change, the *GlobalVersion* is incremented and assigned to S.

Example: Assume that the developer adds a method *multiply* to *BC*, with a *feature* annotation for the *feature MULT*. `MapAssetToFeature` creates this mapping in the AT. The *presencecondition* of the method becomes "*MULT* | `true`".

`CloneAsset` : *asset* $\times$ *asset* $\rightarrow \mathbb{B}$

Description: `CloneAsset` imitates the actual clone&own strategy; when an *asset* is cloned to another location by a developer, `CloneAsset` creates a *deep* clone of the source *asset* and adds it to the target *asset* in the AT, provided it is *containable*. Additionally, if the cloned *asset* (or its sub-assets) is mapped to any features, they are also cloned, added to the target *feature model*, and mapped to the *asset* clone. The clone retains the *version* of the original asset, however, since the target *asset* is modified (addition of sub-asset), the *GlobalVersion* is incremented and assigned to the target. For storing trace links, it creates traces for both *asset* and *feature* clones and adds them to the *TraceDatabase*.

Example: Starting from Fig. 6, the developer copies the method *divide* in *Arithmetic.js*; a file in another project, *ScientificCalculator (SC)*. `CloneAsset` clones *divide* to *Arithmetic.js*, an *asset* of *FileType* in *SC*, as well as the mapped *feature DIV* in the *feature model* of *SC*. Traces for both *divide* and *DIV* are added to the *TraceDatabase*. Figure 7 illustrates the scenario.

`PropagateToAsset` : *asset* $\times$ *asset* $\rightarrow \mathbb{B}$

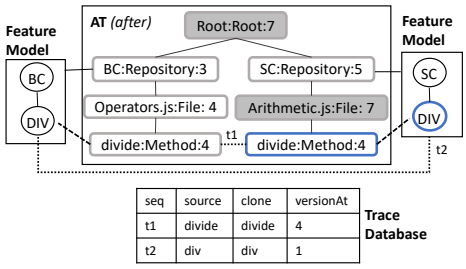

Fig. 7. Illustration of `CloneAsset`(*divide*, *Arithmetic.js*)

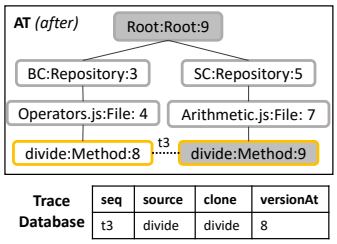

Fig. 8. Illustration of `PropagateToAsset`(*divide*, *divide*)

Description: `PropagateToAsset` takes two assets, checks if one is a clone of the other, and propagates changes in source, after cloning, to its clone. To determine if source was changed, it compares the *version* of source to its *version* when it was cloned (*versionAt* from the *TraceDatabase*). If it is ahead of the *version* it was cloned at, the changes are propagated to the clone. Changes performed in the clone are retained. Propagation, like cloning, includes added and modified sub-assets, added mappings, and renaming. After propagation, a trace with source and clone is added to the *TraceDatabase*, the *versionAt* of which is the *version* of the source. The *GlobalVersion* is incremented and assigned to the clone.

Example: Assume that the *divide* method during cloning did not include the check for division by zero. After adding the check (`ChangeAsset`), the *divide* method in source (*Operators.js*) is ahead (*version*=8) of the *divide* method in target (*Arithmetic.js*), with *version*=4. By invoking `PropagateToAsset`, the changes are propagated automatically. Figure 8 demonstrates the scenario; for simplicity, *feature* mappings are omitted.

B. Feature-Oriented Operators

The feature-oriented operators incorporate feature-related information to the AT and enable feature reuse and maintenance.

`AddFeature` : *feature* $\times$ *feature* $\rightarrow \mathbb{B}$

Description: When a developer adds a *feature* (e.g., in a text file or a database), or an *asset* mapping to a *feature* which does not exist in the *feature model*, `AddFeature` creates a new *feature* and adds it to the *feature model*. It also adds any assets mapped to the *feature* using `AddAsset`. Similar to versioning of `AddAsset`in AT, `AddFeature` increments the *GlobalVersion* (*version* of *root feature*) and assigns it to the added *feature*.

Example: Assume that the *feature model* for *BC* is a textual file, where features are written as individual lines, and indentation is used to represent hierarchy (Clafer syntax [60]). The developer adds a line "EXP" (exponent), below the line "BC" (*root* feature, *BC*). `AddFeature` creates a corresponding *feature EXP*, and adds it to the *feature BC*. The *version* of *root feature* is incremented (previously 1 after adding *feature DIV*) and assigned to *feature EXP*. Figure 9 demonstrates the scenario, with the resulting versions in a table on the right.

`AddFeatureModelToAsset:` *asset* $\times$ *feature model* $\rightarrow \mathbb{B}$

Description: Developers can add a *feature model* to an *asset* in different ways, e.g., as a file or a database. The virtual platform, upon recognizing that a *feature model* is added to an asset in the repository, invokes `AddFeatureModelToAsset`. The operator then locates the *asset* in the AT, creates a *feature model* FM, and sets the asset's parameter *feature*

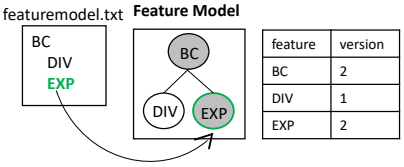

Fig. 9. Illustration of `AddFeature`(*EXP*, *BC*)

model to FM. The *GlobalVersion* of the AT is incremented and assigned to the *asset* which contains FM.

Example: Consider that the *feature model* of *BC* is a separate text file, which resides in the root folder of *BC*. As a result of `AddFeatureModelToAsset`, the *feature model* (FM) will be loaded from the file and assigned to *BC*. All sub-assets of *BC* can now be mapped to features from FM.

`RemoveFeature` : *feature* $\rightarrow \mathbb{B}$

Description: When a *feature* is removed by a developer from a repository, `RemoveFeature` locates the *feature* in the *feature model*, un-maps it from all assets it maps to, and removes the *feature* along with all its sub-features. Additionally, any asset mapped to *only* the removed feature is also removed by the operator. The operator increments the *GlobalVersion* of the FM and assigns it to the parent *feature* (before removal).

`MoveFeature` : *feature* $\times$ *feature* $\rightarrow \mathbb{B}$

Description: Features can be moved in the same project as a result of refactoring, and also across projects, when developers incorporate it into another project. `MoveFeature` combines two operators; `CloneFeature` (explained below) to clone the *feature* (and its mapped assets) to its new location, and `RemoveFeature` to remove it from its previous location.

`MakeFeatureOptional` : *feature* $\rightarrow \mathbb{B}$

Description: Often, developers want to keep a feature's implementation in the AT, and decide whether to include it or not at compile time, instead of deleting it altogether. `MakeFeatureOptional` sets a *feature*'s `boolean` property *optional* to `true`. By default, every *feature* is mandatory when added to the *feature model*. This operator allows to keep the feature's implementation in the AT while allowing developers to activate or deactivate the feature.

`CloneFeature` : *feature* $\times$ *feature* $\rightarrow \mathbb{B}$

Description: Cloning a *feature* manually requires developers to recollect its location in software assets. These assets can be of different types (directory, document, code artifact, text etc). Features can be scattered and therefore harder to locate. This is where the stored (and maintained) meta-data pays off. `CloneFeature` simply invokes a *convenience* operator; `getMappedAssets`, to retrieve all assets mapped to the feature. It then clones the *feature* and all its mapped assets in the target AT and FM. The operator also stores traces for the *asset* and *feature* clones in the *TraceDatabase*. The *GlobalVersion* of the FM is incremented and assigned to the target *feature* (*parent* of the *feature* clone).

Example: After adding the *feature EXP* (using `AddFeature`), the developer added two assets in *feature BC*, and later mapped them to *feature EXP*. The assets are a method "*exponent*" and a textual file "*exp.txt*" with documentation of *exponent*. The developer now wants to reuse *feature EXP* in *SC*. To clone the feature, she invokes `CloneFeature`, which clones the *feature EXP* and its mapped assets to *SC*. Additionally, traces for the *feature* and asset clones are added to the *TraceDatabase*. This example is illustrated in Figure 10. Note that even though *Operators.js* was not cloned, the virtual platform created a clone, as the method *exponent* could not be added directly to the repository. This is referred to as *tree slicing*, which the virtual platform adopts to ensure that the *well-formedness* of the AT is maintained.

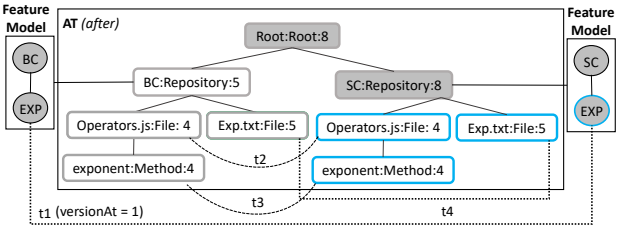

Fig. 10. Illustration of `CloneFeature`(*EXP*, *SC*)

`PropagateToFeature` : *feature* $\times$ *feature* $\rightarrow \mathbb{B}$

Description: `PropagateToFeature` replicates the changes in the *feature* (e.g., renaming, adding or removing sub-features) to either selected, or all of its clones. For checking if propagation is valid and necessary, it checks two conditions, based on the *TraceDatabase*. First, if one of the features provided is a clone of the other. Second, if the *feature* was modified after cloning (current *version* > *versionAt*). After propagating changes, it creates new traces between the source and newly modified targets (both *feature* and *asset*), and adds them to the *TraceDatabase*.

VI. Prototyping and Evaluation

We prototyped and evaluated the virtual platform qualitatively and quantitatively: (i) in a comparative assessment against the frameworks presented in Sec. III, (ii) using a simulation study based on revision histories from clone&own-based system. All results to replicate the evaluation are in our appendix [38]. The prototype, implemented in Scala, provides an API as the main interface to execute the operators. In the production-ready tool, this API would be usable as a command line interface or a set of IDE commands. We used a strategic programming library (kiama) for efficient tree traversal and rewriting. After implementing all operators, we created test scenarios to verify the correctness. These test scenarios were developed using domain knowledge acquired by experience, and also inspired by observing scenarios from the case study of Clafer Web Tools. We checked correctness by comparing the result state (AT, trace, and mappings) after operator invocations to the expected one. We also simulated the illustrative example presented in Sec. II-A by automatically realizing all the discussed scenarios.

A. Comparative Evaluation

For comparison, we extracted activities supported by techniques for supporting clone&own (a.k.a., clone management), or the migration of cloned variants to an integrated platform (a.k.a., product-line migration). In total, we extracted 12 activities which we found to be common across most, if not all, existing

TABLE I
COMPARISON OF THE VIRTUAL PLATFORM WITH ACTIVITIES SUPPORTED BY CLONE-MANAGEMENT AND PRODUCT-LINE MIGRATION FRAMEWORKS

Feature identification → abstract operator [51], specified in the beginning [52], [54], [53], specified any time in virtual platform
Feature location → abstract operator [51], extracted [52], [53], internal tagging [54], also internal tagging in virtual platform
Feature dependency management → abstract operator [51], statically mined [53], specified in beginning [54], specified any time in virtual platform
Feature model creation → multiple abstract operators [51], activity [53], specified in the beginning [54], dynamically grows in virtual platform
Feature-to-asset mapping → abstract operator [51], extracted [52], [53], specified any time [54], specified any time in virtual platform
Clone detection → textual diff tools [51], feature expression comparison [54], git clone points to source [55], not needed in virtual platform
Feature cloning → supported by virtual platform
Change propagation → multiple abstract operators [51], variant synchronization [54], using Git merge [55], automated in virtual platform
Reusable assets creation → abstract & incremental [51], reuse existing variants [52], reusable core assets[53], [55] and features in virtual platform
Product derivation → abstract [51], customizing after cherry-picking [55], composition [52], [53], preprocessor-like in virtual platform
Integration → abstract operator using meta-data [51], third party tool [54], Git merge [55], manual or tool-based, guided by meta-data in the virtual platform
Variant synchronization → Git dif [55], code comparison [52], [53], not needed in virtual platform

techniques. We evaluated the virtual platform's ability to support the scenario from Sec. II and the 12 activities of related frameworks. Details are in the appendix [38].

Table I shows whether and how an activity related to either clone management or product-line migration is supported by an existing framework, as well as virtual platform. The activities are: feature identification (features defined in a variant), feature location (recovering traceability between features and assets), feature dependency management (managing constraints among features), feature model creation (creating and evolving a feature model), storing feature-to-asset mappings, clone detection (identifying assets which are clones of one another), feature cloning (ability to clone features), change propagation (replicating changes made in an asset to its clone), creation of reusable assets (which can be used to derive variants), product derivation (ability to derive a partial or complete product given a configuration), variant integration (merging assets/variants by taking variability into account), and variant comparison (comparison of assets to find commonalities and varaibilities).

In summary, among all frameworks, the virtual platform is the first one fully committed to recording traceability, instead of recovering it later. It automatically maintains traces between cloned assets, and encourages developers to map features to assets of all types and all granularity levels (not just code blocks). This traceability has a cost to developers; however, at the same time, it can significantly reduce cost when complex evolution activities are performed, as detailed below.

The other frameworks define their involved activities either abstractly or using heuristics (e.g., feature location). The virtual platform includes exact specifications and implementations of operators—possible since we address a broad range of evolution scenarios, rather than just the "big bang" scenario of platform migration. The existing methods have not been applied to real project revision histories as part of their evaluation, rather explain that they support migration scenarios described before.

B. Simulation Study

We used an open-source system called Clafer Web Tools (CWT, [61]) that was evolved using clone&own in three cloned variants (*ClaferMooVisualizer*, *ClaferConfigurator*, *ClaferIDE*) towards an integrated platform (*ClaferUICommonPlatform*), including many feature clonings across the variants. We evaluated the virtual platform's efficiency by simulating the evolution of CWT, retrofitting our operators to achieve the original evolution, and studying the costs and benefits.

We used a dataset by Ji et al. [58] that augments the original codebase with feature information, as if it had been developed in a feature-oriented way. It comprises a full revision history for the four sub-systems, with source code from the original developers, and feature information manually added by researchers. Feature information is contained in three types of artifacts: feature models, feature-to-asset mapping files, and embedded feature annotations in code. We provide details about the dataset in our appendix [38].

Performing the Simulation. We retrofitted CWT's full revision history to our operators to extract a sequence of (high-level) operator applications that accurately capture the changes previously expressed by the history of (low-level) file-based commits. We analyzed each pair of successive commits to extract a set of operator applications that produces the delta between the commits. Replaying the operator applications in the given order creates and updates the AT.

Cost & Benefit. As costs, we measure the additional effort imposed on developers by our platform. Our traditional, asset-oriented operators (left-hand column of Table II) do not lead to additional cost, because these tasks are performed in traditional development as well. Cost arises from two components, both related to our feature-oriented operators (right-hand column of Table II): one called C_{feat} for maintaining features, one called C_{miss} for dealing with omissions during feature maintenance. The latter arises if the developer forgets to invoke a feature-oriented operator and then later the feature information is missing for a relevant feature-oriented activity.

As benefits, we consider the saved cost in two dimensions: feature location and clone detection. Feature location cost C_{loc} is saved on invocations of certain operators that rely on previously specified mappings. Clone detection cost C_{clone} is saved on invocations of one certain operator for propagating changes along clones from our clone database.

We study these costs and benefits in four dedicated research questions. RQ1 and RQ2 are devoted to costs, while RQ3 and RQ4 are devoted to benefits. We first discuss these research questions, before weighing off the observed costs and benefits.

RQ1. *What are the costs of maintaining features using feature-oriented operators?* The overall cost C_{feat} arises from accumulating the cost of applying feature-oriented operators. Each feature-oriented operator op has a cost $C_{feat}(op) = \#invoc(op) * cost_{abs}(op)$, which depends on the number of invocations of op, and the absolute cost of each invocation of op. Based on Table II, there are 724 invocations of

TABLE II
OPERATOR INVOCATIONS IN SIMULATION STUDY:ASSET-ORIENTED AND FEATURE-ORIENTED OPERATORS

operator	freq.	operator	freq.
AddAsset	3,527	AddFeature	229
ChangeAsset	1,191	AddFeatureModelToAsset	4
RemoveAsset	1,060	MapAssetToFeature	368
MoveAsset	303	RemoveFeature	40
CloneAsset	48	MoveFeature	22
PropagateToAsset	8	CloneFeature	54
		PropagateToFeature	7

feature-oriented operators in total. Two operators contribute the bulk to this number, namely `MapAssetToFeature` (368) and `AddFeature` (229). The absolute cost per invocation can be assumed to be low (in the order of seconds) because it mostly amounts to picking the feature name, when it is fresh in the developer's mind. An exception are situations where the developer has to deal with earlier omissions (see RQ2).

RQ2. *What percentage of feature maintenance operations required additional feature location effort?* The omission-related cost C_{miss} arises from the number of late invocations of `MapAssetToFeature`, representing situations where the developer missed to specify an asset-to-feature mapping when the asset was added. This number is to be multiplied by the absolute cost for these invocations, which is generally higher than a regular invocation. Our operators `CloneFeature`, and `PropagateToFeature` rely on a complete mapping from a feature to its assets. A third relevant operator is `AddFeature` which adds feature information to source code added earlier. In absence of a recorded mapping, each operator requires an expensive manual feature location step, which is not required in our approach (see RQ3). We counted the number mappings that were added before or after one of these operators was invoked, which indicates that the researcher preparing the original dataset noticed an omission. We determined 14 relevant mappings for `CloneFeature` (2 relevant invocations, 3.7% overall). and 25 relevant mappings for `AddFeature` (12 relevant invocations, 4.0% overall). We did not discover any relevant mappings for `PropagateToFeature`, yielding 39 late invocations in total.

RQ3. *To what extent can feature location costs be avoided when using feature-oriented operators?* The operators `CloneFeature` and `PropagateFeature` rely on previously specified mappings. Conversely to RQ2, we can assume that each invocation of one of these operators avoided manual feature location when it did not require any fixing of omitted annotations. So, we define C_{loc} to rely on the number of feature location steps saved by an invocation of one of our operators. We count 54 invocations of `CloneFeature`, and 7 relevant invocations of `PropagateToFeature`, leading to a final value of 61. This number is to be multiplied with the absolute cost of feature location, which can be assumed to be high (earlier work [58] gives an estimate of 15 minutes per feature), based a strong reliance on the developers' memory, and an understanding of how cross-cutting features are scattered.

RQ4. *To what extent can clone detection costs be avoided when using feature-oriented operators?* Since the propagation of changes along clones requires a complete specification of the clones at hand, we can assume that every application of `PropagateToFeature` saves one application of clone detection (either manual or using a tool). In our subject system, we identified 7 invocations of `PropagateToFeature`. To obtain the value of C_{clone}, this number of is to be multiple with the absolute cost for clone detection. Manual clone detection is a tedious and error-prone task, and known to be infeasible for larger systems [62]. Tool-based clone detection requires manual verification and postprocessing, since even the most advanced clone detection tools have imperfect precision and recall [63].

C. Discussion

Break-Even Point. We can now weigh off the costs observed in RQ1+2 against the benefits from RQ3+4. Consider the following formula, which specifies the total benefit of using the virtual platform: B_{total} = -(C_{feat} + C_{miss}) + (C_{loc} + C_{clone}). If this formula yields a positive value, the virtual platform surpasses the break-even point and leads to a net benefit.

The value of B_{total} depends on the absolute costs for operator invocations, feature location, and clone detection, which are unavailable. However, we can perform an approximation based on plausible estimates: (1.) For the cost of feature location, we rely on the earlier literature estimate [58] of 15 minutes per instance. (2.) We assume clone detection to have the same cost as feature location. (3.) We assume the cost for adding an omitted annotation to be 10 times as high as a regular operator invocation. Based on these three assumptions, we break even if *invoking a feature-oriented operator takes 54 seconds or less* on average. In practice, the benefit can be assumed to be larger, since invoking a feature-oriented operator mostly entails picking a feature name (while the feature is still fresh in the developer's mind), a matter of a few seconds.

This calculation shows promising results in terms of saved effort and time. By simulating the development of the case study with feature-oriented information, we can reuse as much as 20 features from one project (*ClaferMooVisualizer*) by cloning them. We envision greater accuracy and efficiency levels when the virtual platform is used alongside development.

Representativeness. Our case is representative for systems of comparable size (547k lines, four variants). Many reported product-line migrations are of similar size [64]. We argue for representativeness for larger systems qualitatively. Our case has all evolution activities observable in industrial systems, supported by other frameworks. Still, the virtual platform is evaluated more extensively than any of these.

Threats to Validity. A threat to external validity is that our operators do not completely capture the real-world scenarios developers encounter when dealing with variant-rich systems. We mitigate this threat with our evaluation based on the simulation of a real system. There is a general lack of available systems for benchmarking on realistic revision histories with available feature information, a problem that we aim to address as part of our ongoing benchmarking initiative [65], [66].

There are two main threats to internal validity. First, our calculation of C_{miss} could be incomplete: there might be potential omissions not fixed by a later commit. This situation is comparable to other research that relies on potentially imperfect datasets (e.g., in software defect prediction [67], [68]). While our analysis focuses on omissions that later required fixing, these omissions are arguably the most relevant ones in practice. Second, there could be implementation errors; after retrofitting our operations to the development process given by the commit revision, the AT might be in an incorrect state. To mitigate this threat, one author, not involved in the simulation, manually inspected a random sample of 25 commits by comparing the git *diff* with the AT resulting from operator invocations. The AT was always consistent.

VII. Related Work

The five most closely related works are the clone-management and product-line-migration frameworks that we used to inform the virtual platform's design ([51], [52], [53], [54], [55], cf. Sec. III). In Sec. VI-A and our online appendix [38], we provide a detailed comparison, highlighting unique benefits of the virtual platform: support for early traceability recording, operators for the full spectrum between the extremes ad hoc clone&own and integrated platform, and an evaluation on a real project revision history. We now discuss further related work on product-line migration and integrated-platform evolution.

The idea of automatically handling variation points, as the virtual platform does, is not new. In fact, going back to the 1970s, researchers have built so-called variation-control systems [69], [70], which never made it into the practice of software engineering. These systems have been realized upon different back- and frontends (e.g., version-control systems [71], [72] or a text editor [73]), but before effective and scalable concepts from SPLE research for managing variability have been established. The virtual platform can be seen as a variation control system.

The large majority of product-line migration techniques focuses on detecting and analyzing commonalities and variabilities of the cloned variants, together with feature identification and location, as shown in Assuncao et al.'s recent mapping study based on 119 papers [74]. Case studies of manual migration [48], [18], [20], [75], [76], [17] also exist. These illustrate the difficulties and huge efforts of recovering important information (features and clone relationships) that was never recorded during clone&own, supporting our approach of recording such information early. Finally, many works focus on migrating a *single system* into a configurable, product-line platform [77], [76], [75], [78], typically proposing refactoring techniques. Wille et al. [79] use variability mining to generate transformational rules for creating delta-oriented product lines.

Others focus on evolving software platforms. Liebig et al. [80] present variability-aware sound refactorings (rename identifier, extract function, inline function) for evolving a platform by preserving the variants. Rabiser et al. [81] present an approach for managing clones at product, component, and feature, and define 5 consistency levels to monitor co-evolving clones. Ignaim et al. [82] present an extractive approach to engineer cloned variants into systematic reuse. Neves et al. [83] propose a set of operators for safe platform evolution. In contrast to our operationally defined operators, these operators are defined on an abstract level, based on their pre- and post-conditions; implementing them is left to the user. Incorporating safe evolution or Morpheus' refacting in the virtual platform is a valuable future work.

VIII. Conclusion

We designed, formalized, and prototyped the virtual platform—a framework that exploits a spectrum between the two extremes ad hoc clone&own and fully integrated platform, supporting both kinds of development. Based on the number of variants, organizations can decide to use only a subset of all the variability concepts typically required for an integrated platform, fostering flexibility and innovation, starting with clone&own and incrementally scaling the development. This realizes incremental benefits for incremental investments and even allows to use clone&own when a platform is already established, to support a more agile development. Another core novelty is that, instead of trying to expensively recover relevant meta-data (e.g., features, feature locations, and clone traces), the virtual platform fosters recording it early. For instance, developers typically know the feature they are implementing, but usually do not record it. The virtual platform records such meta-data and exploits it for the transition, providing operators that developers can use to handle variability. Our evaluation shows that the additional costs are low compared to the benefits.

We see several promising directions of future work. By allowing developers to continuously record feature meta-data, the virtual platform paves the way for software analyses that rely on this data. One example is support for the safe evolution of product line platforms [83], which could be extended to support systems in our intermediate governance levels. Specifying our operators in the framework of software product line transformations [84], [85], [86] would make them amenable to conflict and dependency analysis [87], a versatile formal analysis with applications in the coordination of evolution processes. Many of the virtual platform's operators (e.g., those related to change propagation) lead to non-trivial changes of the codebase. To increase developer trust and optimize accuracy, an important challenge is to keep the "human in the loop", which we aim to address by exploring dedicated user interfaces. By integrating the virtual platform with available annotation systems [88], we could facilitate inspection of the available feature mappings. Offering a "preview mode" would allow to inspect and interact with the changes arising from a planned operator invocation. Providing a dedicated operator to integrate cloned features is another future direction. Other directions are to support configuration of variants by selecting features, offering views [89], and providing visualizations (e.g., dashboards [90], [91]). Finally, recommender systems that learn from the meta-data and support developers handling features and assets could further encourage using features in software engineering [92].

Acknowledgment. Swedish Research Council (257822902), Vinnova Sweden (2016-02804), and the Wallenberg Academy.

REFERENCES

[1] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An exploratory study of cloning in industrial software product lines," in *CSMR*, 2013.

[2] Ş. Stănciulescu, S. Schulze, and A. Wąsowski, "Forked and integrated variants in an open-source firmware project," in *ICSME*, 2015.

[3] J. Businge, O. Moses, S. Nadi, E. Bainomugisha, and T. Berger, "Clone-based variability management in the android ecosystem," in *ICSME*, 2018.

[4] J. Krueger and T. Berger, "An empirical analysis of the costs of clone- and platform-oriented software reuse," in *FSE*, 2020.

[5] N. Lodewijks, "Analysis of a clone-and-own industrial automation system: An exploratory study," in *SATToSE*, 2017.

[6] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wasowski, "A survey of variability modeling in industrial practice," in *VaMoS*, 2013.

[7] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, 2001.

[8] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, 2000.

[9] F. J. van der Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*, 2007.

[10] S. Apel, D. Batory, C. Kästner, and G. Saake, in *Feature-Oriented Software Product Lines*. Springer, 2013.

[11] T. Berger, J.-P. Steghöfer, T. Ziadi, J. Robin, and J. Martinez, "The state of adoption and the challenges of systematic variability management in industry," *Empirical Software Engineering*, vol. 25, no. 3, pp. 1755–1797, 2020.

[12] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Carnegie-Mellon University, Pittsburgh, PA, USA, Tech. Rep., 1990.

[13] D. Nesic, J. Krueger, S. Stanciulescu, and T. Berger, "Principles of feature modeling," in *FSE*, 2019.

[14] T. Berger, S. She, R. Lotufo, K. Czarnecki, and A. Wasowski, "Feature-to-code mapping in two large product lines," in *SPLC*, 2010, extended Abstract.

[15] L. Linsbauer, E. R. Lopez-Herrejon, and A. Egyed, "Recovering traceability between features and code in product variants," in *SPLC*, 2013.

[16] R. Bashroush, M. Garba, R. Rabiser, I. Groher, and G. Botterweck, "Case tool support for variability management in software product lines," *ACM Computing Surveys (CSUR)*, vol. 50, no. 1, pp. 1–45, 2017.

[17] F. Stallinger, R. Neumann, R. Schossleitner, and S. Kriener, "Migrating towards evolving software product lines: Challenges of an SME in a core customer-driven industrial systems engineering context," in *PLEASE*, 2011.

[18] H. P. Jepsen, J. G. Dall, and D. Beuche, "Minimally invasive migration to software product lines," in *SPLC*, 2007.

[19] W. K. G. Assunção, R. E. Lopez-Herrejon, L. Linsbauer, S. R. Vergilio, and A. Egyed, "Reengineering legacy applications into software product lines: a systematic mapping," *Empirical Software Engineering*, vol. 22, no. 6, pp. 2972–3016, 2017.

[20] J. Krueger and T. Berger, "Activities and costs of re-engineering cloned variants into an integrated platform," in *VaMoS*, 2020.

[21] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Name suggestions during feature identification: The variclouds approach," in *SPLC*, 2016.

[22] S. Zhou, S. Stănciulescu, O. Leßenich, Y. Xiong, A. Wasowski, and C. Kästner, "Identifying features in forks," in *ICSE*, 2018.

[23] S. B. Nasr, G. Bécan, M. Acher, J. B. F. Filho, N. Sannier, B. Baudry, and J. Davril, "Automated extraction of product comparison matrices from informal product descriptions," *Journal of Systems and Software*, vol. 124, pp. 82–103, 2017.

[24] J. Rubin and M. Chechik, "A Survey of Feature Location Techniques," in *Domain Engineering*, 2013, pp. 29–58.

[25] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[26] G. K. Michelon, L. Linsbauer, W. K. Assunção, S. Fischer, and A. Egyed, "A hybrid feature location technique for re-engineering single systems into software product lines," in *VaMoS*, 2021.

[27] C. Kästner, A. Dreiling, and K. Ostermann, "Variability mining with leadt," *Tec. Rep., Philipps Univ. Marburg*, 2011.

[28] C. Kästner, A. Dreiling, and K. Ostermann, "Variability mining: Consistent semi-automatic detection of product-line features," *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 67–82, 2013.

[29] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queen's School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[30] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[31] J. Wang, X. Peng, Z. Xing, and W. Zhao, "How developers perform feature location tasks: a human-centric and process-oriented exploratory study," *Journal of Software: Evolution and Process*, vol. 25, no. 11, pp. 1193–1224, 2013.

[32] S. Grüner, A. Burger, T. Kantonen, and J. Rückert, "Incremental migration to software product line engineering," in *SPLC*, 2020.

[33] J. Krüger, W. Mahmood, and T. Berger, "Promote-pl: a round-trip engineering process model for adopting and evolving product lines," in *SPLC*, 2020.

[34] B. Zhang, M. Becker, T. Patzke, K. Sierszecki, and J. E. Savolainen, "Variability evolution and erosion in industrial product lines: a case study," in *SPLC*, 2013.

[35] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, t. Stănciulescu, A. Wasowski, and I. Schaefer, "Flexible product line engineering with a virtual platform," in *ICSE-NIER*, 2014.

[36] T. Fogdal, H. Scherrebeck, J. Kuusela, M. Becker, and B. Zhang, "Ten years of product line engineering at danfoss: lessons learned and way ahead," in *SPLC*, 2016.

[37] "Virtual platform prototype," https://bitbucket.org/easelab/workspace/projects/VP.

[38] "Appendix," https://bitbucket.org/easelab/2021-icse-vponlineappendix.

[39] M. A. Laguna and Y. Crespo, "A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring," *Science of Computer Programming*, vol. 78, no. 8, pp. 1010–1034, 2013.

[40] L. Passos, R. Queiroz, M. Mukelabai, T. Berger, S. Apel, K. Czarnecki, and J. Padilla, "A study of feature scattering in the linux kernel," *IEEE Transactions on Software Engineering*, vol. 47, pp. 146–164, 2021.

[41] L. Passos, J. Padilla, T. Berger, S. Apel, K. Czarnecki, and M. T. Valente, "Feature scattering in the large: A longitudinal study of Linux kernel device drivers," in *MODULARITY*, 2015.

[42] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki, "What is a feature? a qualitative study of features in industrial software product lines," in *SPLC*, 2015.

[43] J. Krüger, W. Gu, H. Shen, M. Mukelabai, R. Hebig, and T. Berger, "Towards a better understanding of software features and their characteristics: a case study of marlin," in *VaMoS*, 2018.

[44] J. Krüger, M. Mukelabai, W. Gu, H. Shen, R. Hebig, and T. Berger, "Where is my feature and what is it about? a case study on recovering feature facets," *Journal of Systems and Software*, vol. 152, pp. 239–253, 2019.

[45] J. Krüger, L. Nell, W. Fenske, G. Saake, and T. Leich, "Finding Lost Features in Cloned Systems," in *SPLC*, 2017.

[46] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, "A study of variability models and languages in the systems software domain," *IEEE Transactions on Software Engineering*, vol. 39, no. 12, pp. 1611–1640, 2013.

[47] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk, "Is The Linux Kernel a Software Product Line?" in *SPLC-OSSPL*, 2007.

[48] W. A. Hetrick, C. W. Krueger, and J. G. Moore, "Incremental return on incremental investment: Engenio's transition to software product line practice," in *OOPSLA*, 2006.

[49] D. Bilic, D. Sundmark, W. Afzal, P. Wallin, A. Causevic, C. Amlinger, and D. Barkah, "Towards a model-driven product line engineering process: An industrial case study," in *ISEC*, 2020.

[50] J. Rubin, K. Czarnecki, and M. Chechik, "Managing cloned variants: A framework and experience," in *SPLC*, 2013.

[51] ——, "Cloned product variants: from ad-hoc to managed software product lines," *International Journal on Software Tools for Technology Transfer*, vol. 5, no. 17, pp. 627–646, 2015.

[52] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Enhancing clone-and-own with systematic reuse for developing software variants," in *ICSME*, 2014.

[53] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Bottom-up technologies for reuse: Automated extractive adoption of software product lines," in *ICSE-C*, 2017.

[54] T. Pfofe, T. Thüm, S. Schulze, W. Fenske, and I. Schaefer, "Synchronizing software variants with variantsync," in *SPLC*, 2016.

[55] L. Montalvillo and O. Díaz, "Tuning github for spl development: branching models & repository operations for product engineers," in *SPLC*, 2015.

[56] S. Apel, C. Kästner, and C. Lengauer, "Featurehouse: Language-independent, automated software composition," in *ICSE*, 2009.

[57] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "The variability model of the linux kernel." *VaMoS*, 2010.

[58] W. Ji, T. Berger, M. Antkiewicz, and K. Czarnecki, "Maintaining feature traceability with embedded annotations," in *SPLC*, 2015.

[59] D. Strüber, A. Anjorin, and T. Berger, "Variability representations in class models: An empirical assessment," in *MODELS*, 2020.

[60] K. Bąk, Z. Diskin, M. Antkiewicz, K. Czarnecki, and A. Wąsowski, "Clafer: unifying class and feature modeling," *Software & Systems Modeling*, vol. 15, no. 3, pp. 811–845, 2016.

[61] M. Antkiewicz, K. Bak, A. Murashkin, R. Olaechea, J. Hui, and K. Czarnecki, "Clafer tools for product line engineering." in *SPLC Workshops*, 2013.

[62] R. Koschke, "Survey of research on software clones," in *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[63] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *ICSE*, 2016.

[64] J. Martinez, W. K. G. Assunção, and T. Ziadi, "Espla: A catalog of extractive spl adoption case studies," in *SPLC*, 2017.

[65] D. Strüber, M. Mukelabai, J. Krüger, S. Fischer, L. Linsbauer, J. Martinez, and T. Berger, "Facing the truth: benchmarking the techniques for the evolution of variant-rich systems," in *SPLC*, 2019.

[66] T. Berger, M. Chechik, T. Kehrer, and M. Wimmer, "Software evolution in time and space: Unifying version and variability management (dagstuhl seminar 19191)," in *Dagstuhl Reports*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2019.

[67] C. Catal and B. Diri, "A systematic review of software fault prediction studies," *Expert systems with applications*, vol. 36, no. 4, pp. 7346–7354, 2009.

[68] S. Strüder, M. Mukelabai, D. Strüber, and T. Berger, "Feature-oriented defect prediction," in *SPLC*, 2020.

[69] L. Linsbauer, T. Berger, and P. Grünbacher, "A classification of variation control systems," in *GPCE*, 2017.

[70] L. Linsbauer, F. Schwaegerl, T. Berger, and P. Gruenbacher, "Concepts of variation control systems," *Journal of Systems and Software*, vol. 171, p. 110796, 2021.

[71] B. P. Munch, J.-O. Larsen, B. Gulla, R. Conradi, and E.-A. Karlsson, "Uniform versioning: The change-oriented model," in *SCM*, 1993.

[72] A. Lie, R. Conradi, T. Didriksen, and E. Karlsson, "Change oriented versioning in a software engineering database," in *SCM*, 1989, pp. 56–65.

[81] D. Rabiser, P. Grünbacher, H. Prähofer, and F. Angerer, "A prototype-based approach for managing clones in clone-and-own product lines," in *Proceedings of the 20th International Systems and Software Product Line Conference*, 2016, pp. 35–44.

[73] V. J. Kruskal, "Managing multi-version programs with an editor," *IBM Journal of Research and Development*, vol. 28, no. 1, pp. 74–81, 1984.

[74] W. K. G. Assunção, R. E. Lopez-Herrejon, L. Linsbauer, S. R. Vergilio, and A. Egyed, "Reengineering legacy applications into software product lines: A systematic mapping," *Empirical Software Engineering*, vol. 22, no. 6, pp. 2972–3016, 2017.

[75] C. Kästner, S. Apel, and D. S. Batory, "A case study implementing features using aspectj," in *SPLC*, 2007.

[76] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi, "Refactoring a legacy component for reuse in a software product line: a case study," *Journal of Software Maintenance*, vol. 18, no. 2, pp. 109–132, 2006.

[77] S. Schulze, T. Thüm, M. Kuhlemann, and G. Saake, "Variant-preserving refactoring in feature-oriented software product lines," in *VaMoS*, 2012.

[78] J. Cleland-Huang, A. Agrawal, M. N. A. Islam, E. Tsai, M. Van Speybroeck, and M. Vierhauser, "Requirements-driven configuration of emergency response missions with small aerial vehicles," in *SPLC*, 2020.

[79] D. Wille, T. Runge, C. Seidl, and S. Schulze, "Extractive software product line engineering using model-based delta module generation," in *VaMoS*, 2017.

[80] J. Liebig, A. Janker, F. Garbe, S. Apel, and C. Lengauer, "Morpheus: Variability-aware refactoring in the wild," in *ICSE*, 2015.

[82] K. Ignaim, J. M. Fernandes, A. L. Ferreira, and J. Seidel, "A systematic reuse-based approach for customized cloned variants," in *QUATIC*, 2018.

[83] L. Neves, P. Borba, V. Alves, L. Turnes, L. Teixeira, D. Sena, and U. Kulesza, "Safe evolution templates for software product lines," *Journal of Systems and Software*, vol. 106, pp. 42–58, 2015.

[84] G. Taentzer, R. Salay, D. Strüber, and M. Chechik, "Transformations of software product lines: A generalizing framework based on category theory," in *MODELS*, 2017.

[85] D. Strüber, S. Peldszus, and J. Jürjens, "Taming multi-variability of software product line transformations," in *FASE*, 2018.

[86] M. Chechik, M. Famelis, R. Salay, and D. Strüber, "Perspectives of model transformation reuse," in *IFM*, 2016.

[87] L. Lambers, D. Strüber, G. Taentzer, K. Born, and J. Huebert, "Multi-granular conflict and dependency analysis in software engineering based on graph transformation," in *ICSE*, 2018.

[88] T. Schwarz, W. Mahmood, and T. Berger, "A common notation and tool support for embedded feature annotations," in *SPLC*, 2020.

[89] S. Stanciulescu, T. Berger, E. Walkingshaw, and A. Wąsowski, "Concepts, operations, and feasibility of a projection-based variation control system," in *ICSME*, 2016.

[90] B. Andam, A. Burger, T. Berger, and M. R. V. Chaudron, "Florida: Feature location dashboard for extracting and visualizing feature traces," in *VaMoS*, 2017.

[91] S. Entekhabi, A. Solback, J.-P. Steghöfer, and T. Berger, "Visualization of feature locations with the tool featuredashboard," in *SPLC, Tools Track*, 2019.

[92] H. Abukwaik, A. Burger, B. Andam, and T. Berger, "Semi-automated feature traceability with embedded annotations," in *ICSME*, 2018.