# Virtual Platform: Effective and Seamless Variability Management for Software Systems

Wardah Mahmood, Gül Çalıklı, Daniel Strüber, Ralf Lämmel, Mukelabai Mukelabai, and Thorsten Berger

**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 and agile, 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 the opposing strategies. We design, formalize, and prototype a variability-management framework: the virtual platform. Virtual platform 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. A cost-and-benefit analysis of using the virtual platform to simulate the development of a real-world variant-rich system shows that it leads to benefits in terms of saved effort and time for clone detection and feature location. Furthermore, we present a user study indicating that the virtual platform effectively supports exploratory and hands-on tasks, outperforming manual development concerning correctness. We also observed that participants were significantly faster when performing typical variability management tasks using the virtual platform. Furthermore, participants perceived manual development to be significantly more difficult than using the virtual platform, preferring virtual platform for all our tasks. We supplement our findings with recommendations on when to use virtual platform and on incorporating the virtual platform in practice.

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

## 1 Introduction

Software systems often need to exist in 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. Variants allow 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], [6], 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 [7], [1], [5] 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) [8], [9], [10], [11], [12], [13]. 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 [14], [15], [16], [17], [18], robotics systems [19], [20], and industrial automation systems [12], [21]) or highly configurable systems (e.g., the Linux kernel [22], [23], [24], [25]). This strategy scales, but is often difficult to adopt and requires substantial up-front investments, since variability concepts (e.g., a feature model [26], [27], feature-to-asset traceability [28], [29], a configuration tool [30]) 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 [31], [32], [7], [33], eliciting meta-data that was never recorded during clone & own, such as features and their locations in software assets [34], [27].

Over the last decades, researchers focused on heuristic techniques to recover information from legacy codebases, including feature identification [35], [36], [37], feature location [38], [39], [40], variability mining [41], [42], and clone-detection techniques [43], [44]. 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


- *W. Mahmood and D. Strüber are with the Department of Computer Science and Engineering, Chalmers | University of Gothenburg, Sweden. E-mails: wardah@chalmers.se; danstru@chalmers.se*
- *G. Çalıklı is with School of Computing Science, University of Glasgow, United Kingdom.*
- *R. Lämmel is with the University of Koblenz-Landau, Germany.*
- *M. Mukelabai was with the Department of Computer Science and Engineering, Chalmers | University of Gothenburg, Sweden when working on the paper, and is now with the Department of Computer Science, University of Zambia, Zambia.*
- *T. Berger is with the Faculty of Computer Science, Ruhr University Bochum, Germany, and with the Department of Computer Science and Engineering, Chalmers | University of Gothenburg, Sweden.*

entry points for feature location techniques [45]). 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 the migration 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, their size, the number and diversity of the implemented features, and the available resources, 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 [46], [12], [47], [48]. 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 [49] 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 [31] often leading to re-engineering efforts over years [32], [21]. 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 is the conceptual structures and formalized operators for the virtual platform, which are related to ordinary code editing, but also record and exploit meta-data.

Our evaluation of the proposed framework is two-fold. First, we prototypically implemented the virtual platform on top of an ordinary file system. Using the prototype, we simulated the development of variants of a medium-sized system (57.4k lines of text, 4 variants). We verified that our prescribed operators sufficed to cover each evolution scenario undertaken in the development of the variants. We also conducted a cost and benefit analysis of using our framework, the results of which indicated that using virtual platform leads to significant cost savings during inevitable evolution and maintenance activities. Second, as an extension to our prototype-based evaluation of the virtual platform, we conducted a user study with 12 participants, where they were required to perform routine developer tasks on two subject systems to assess the effect on developer performance using the virtual platform, in terms of correctness and efficiency measures. In addition, we present a qualitative analysis over the subjective preferences that we retrieved from our user study participants. Our results show that virtual platform outperformed manual mode of work for all tasks with respect to correctness. Participants also performed the tasks faster when using the virtual platform, significantly so for all tasks. Additionally, in terms of subjective preferences, participants perceived manual mode of work to be more difficult than using the virtual platform for all tasks, and subsequently indicated their preference to use virtual platform over manual mode of work for all tasks.

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 (Sections 6 and 7),
- **a prototype of the virtual platform [50]** in Scala,
- **a command-line interface for virtual platform** that allows invoking the operators using a common modality,
- **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,
- a **user study** with 12 participants to measure the accuracy and efficiency of performing routine software evolution tasks using the virtual platform, and
- **an online appendix [51]** with a technical report about our operators, additional examples, data from the simulation study, and a study replication package.

This paper significantly extends our earlier conference paper [52], where we present a conceptualization and initial evaluation of virtual platform. In the present work, we make the following extensions. First, to support the interactions of developers with the virtual platform, we developed a command-line interface. Second, we extended our language supporting towards Java [53] by developing a Java parser (which mimics typical Java parsers, but can also parse Java code with *embedded feature annotations* (e.g., preprocessor directives) that record features in code. Third, we implemented various technical improvements: a new, more robust versioning scheme, persistence over traceability metadata, serialization (to persist in-memory modifications of conceptual structures; previously only a prototypical implementation for the evaluation existed), and code-level cloning and change propagation. Fourth, we conducted a user study with 12 participants, comparing the correctness and efficiency of frequent developer tasks manually and using the virtual platform. Fifth, we extended our discussion of findings with practical implications and recommendations in Sec. 10.5.

## 2 Motivation and Overview

We provide a 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 migrations [33], [54]. It includes tasks that are tedious and error-prone in practice (e.g., bug-fix propagation along branches). Notably, while we present the details and technicalities of our framework only later (Sec. 6 and Sec. 7), we already introduce some of the concepts intuitively using examples and scenarios presented below for a smooth flow and easier comprehension. Additionally, we present the problems and solutions in the same section (instead of having a separate section for the latter) for cohesion and conciseness.

### 2.1 Motivating Running Example

We now discuss relevant problems of managing variants inspired by industrial practices, also presenting our solution in the virtual platform and how a developer would use it. Specifically, developers interact with the virtual platform by invoking its provided operators, either via the command-line interface or an integration with an IDE or version-control system provided by a tool vendor (see Sec. 4 for details). While the traditional, asset-oriented operators (e.g., copying an asset using `CloneAsset`, explained in Sec. 7.1) can run transparently in the background, only the feature-oriented operators (e.g., mapping an asset to a feature using `MapAssetToFeature`, explained in Sec. 7.2) require an extra user interaction for invoking the operators. The operator are described in detail in Sec. 7.

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.

Fig. 1 illustrates the two opposing strategies (cf. Sec. 1) 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 [55], [56]), as well as the representation of variability in the integrated platform. We also need more variability concepts, among others, features [57], [58], [59], code-level configuration [11], feature-to-asset traceability [28], [29], [60], a feature model (a hierarchical structure with features and their dependencies) [26], [27], a configurable build system [11], and a configurator tool [30], [23], [61]. 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,

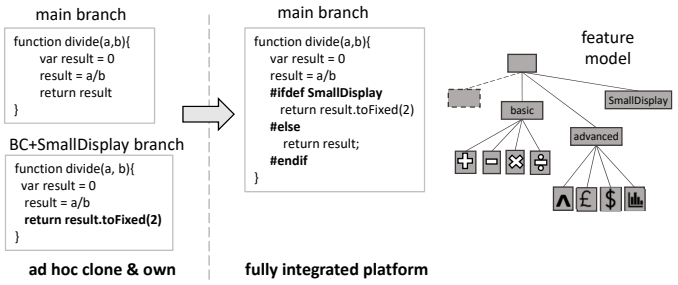



Fig. 1: Ad hoc clone & own vs. fully integrated platform illustrated for two variants: the *BasicCalculator* and a variant with only a small display

migration can be invasive, risky, and costly, especially hard to achieve under market pressure [62], [32], [21], [34], [31], [63].

The virtual platform exploits a spectrum between the two extremes and supports an incremental transition as in Fig. 2. It adapts the governance levels from prior work [49], 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, Level 1). 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)

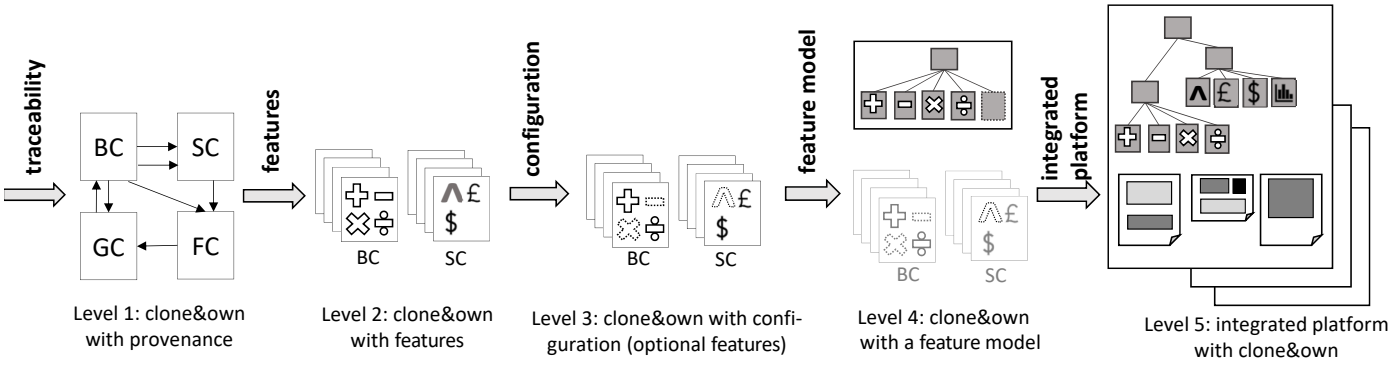



Fig. 2: Spectrum between the extremes ad hoc clone & own and a fully integrated platform (see Fig. 1 for both), illustrated with cloned variants: *BasicCalculator* (BC), *ScientificCalculator* (SC), *GraphingCalculator* (GC), and *FinancialCalculator (FC)*.

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, Level 2). 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 [64]. The virtual platform offers operators to add features conveniently (which then automatically map the related assets to the features).

The developer maps related 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?** Even though there are features, which help maintaining variants, substantial redundancy still exists.

**Solution 3: Clone & own with configuration** (Fig. 2, Level 3). To reduce redundancy, 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, Level 4). 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 by adding and deleting features at any 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, Level 5). Our organization integrates the projects into a consolidated platform. Conveniently, 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.

## 3 Related Work

We now present the existing frameworks for clone-management and product-line-migration that informed the virtual platform's design (cf. Sec. 5). We also present the related techniques on product-line migration and evolution.

### 3.1 Related Frameworks and Detailed Differences

We identified seven relevant works on clone management and product-line migration using our experience and knowledge of literature. Rubin et al.'s product-line migration framework [65], [66] offers operators that support the narrative that a mechanization—i.e., an operator-based perspective—leads to more efficient implementation and support. They propose abstract operators for mining the metadata pertaining to features, their dependencies, and their implementation in assets, which is used to create a tentative architecture and feature model. Fischer et al.'s [67] 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 [68] is also a heuristic-based extraction technique for product-line migration, including support for feature-model synthesis. Pfofe et al.'s tool VariantSync [69] supports clone management by easing the synchronization of assets among cloned variants. Their tool is the only other work that relies on proactively added metadata about features and their implementation. Montalvillo et al.'s operators and branching models for clone management in version-control systems [70] allow isolated variant development with change propagation. König et al. [71] provide a heuristic-based technique for automated change propagation in asset clones across two dimensions: intra-clone (clones withing variants) and inter-clone (clones across variants). Schmorleiz and Lämmel [72] present a process for handling the similarity between different cloned variants of any software system, and automatically propagating changes between them.

Notably, virtual platform, like Rubin et al's framework [65], [66], also offers an operator-based perspective. However, in contrast to their framework, it provides concrete operators that can be invoked by developers to perform various tasks. Additionally, all frameworks for product-line migration [65], [66], [67], [68] rely on heuristics (e.g., code similarity) for identifying clones and locating features. Virtual platform is the first framework to rely on proactively recorded metadata pertaining to clone traceability and feature-to-asset traceability to achieve better accuracy and consistency. Pfofe et al.'s tool VariantSync [69], although uses the notion of features and feature-to-asset mapping for change propagation (a.k.a variant synchronization), it only caters for code-level assets. Virtual platform allows mappings and change propagation in assets at higher level of granularity, such as repository, folder, and files. Lastly, each of the studied change propagation frameworks [70], [71], [72], while accurate, do not cater for features, and therefore do not allow feature change propagation, a common occurrence in feature-oriented software development.

For a deeper comparison, we extracted activities supported by the above-mentioned frameworks for supporting clone management or migration of cloned variants to an integrated platform (a.k.a. product-line migration). For brevity, we only briefly summarize the identified activities here. Detailed descriptions are in our online appendix [51]. In total, we extracted 12 activities we found to be common across most, if not all, existing techniques. We evaluated the frameworks based on their ability to support the scenarios from Sec. 2 which are captured in the 12 activities we extracted. Details are in the appendix [51].

TABLE 1: Comparison of virtual platform with activities from clone-management and product-line migration frameworks

**Feature identification** → abstract operator [66], specified in the beginning [67], [69], [68], specified any time in virtual platform
**Feature location** → abstract operator [66], extracted [67], [68], code-level tagging [69], code- and non-code-level tagging in virtual platform
**Feature dependency management** → abstract operator [66], statically mined [68], specified in the beginning [69], specified any time in virtual platform
**Feature model creation** → multiple abstract operators [66], activity [68], specified in the beginning [69], dynamically grows in virtual platform
**Feature-to-asset mapping** → abstract operator [66], extracted [67], [68], specified at any time in [69] and virtual platform
**Clone detection** → textual diff tools [66], feature expression comparison [69], git clone points to source [70], heuristic-based AST similarity [71], [72], not needed in virtual platform
**Feature cloning** → supported by virtual platform
**Change propagation** → multiple abstract operators [66], variant synchronization [69], using Git merge [70], revision-based and model-based [71], automated in [72] and virtual platform
**Reusable assets creation** → abstract & incremental [66], reuse existing variants [67], reusable core assets[68], [70], reusable assets and features in virtual platform
**Product derivation** → abstract [66], customizing after cherry-picking [70], composition [67], [68], preprocessor-like in [69] and virtual platform
**Integration** → abstract operator using meta-data [66], not specified in [69], Git merge [70], manual or tool-based, guided by meta-data in the virtual platform
**Variant synchronization** → Git diff [70], AST comparison [71], code comparison [67], [68], not needed in virtual platform

Table 1 shows if and how an activity related to either clone management or product-line migration is supported by an existing framework, as well as the 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 variabilities).

In summary, the existing frameworks define their activities either abstractly or using heuristics (e.g., for feature location). In contrast, the virtual platform includes exact specifications and implementations of operators, which allows addressing a broad range of evolution scenarios rather than just the "big bang" scenario of platform migration. Additionally, the virtual platform is the first framework that is fully committed to recording traceability at all levels of granularity (repository, folder, file, code) instead of recovering it later. This traceability has a cost to developers; however, at the

same time, it can significantly reduce cost when complex evolution activities are performed, as shown in Sec. 9. Virtual platform is also the first framework to provide support for both variant management and product-line migration. Lastly, the existing methods have not been applied to real project revision histories as part of their evaluation, and instead, only explain that they support migration scenarios described before. More details can be found in our online appendix [51].

### 3.2 Product-Line Evolution and Migration

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 [73], [74], which never made it into the practice of software engineering. These systems have been realized upon different back- and front-ends (e.g., version-control systems [75], [76] or a text editor [77]), 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 [78]. Case studies of manual migration [62], [32], [34], [79], [80], [31] 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 [81], [80], [79], [82], typically proposing refactoring techniques. Wille et al. [83] use variability mining to generate transformational rules for creating delta-oriented product lines.

Others focus on evolving software platforms. Liebig et al. [84] present variability-aware sound refactorings (rename identifier, extract function, inline function) for evolving a platform by preserving the variants. Rabiser et al. [85] present an approach for managing clones at product, component, and feature, and define 5 consistency levels to monitor co-evolving clones. Ignaim et al. [86] present an extractive approach to engineer cloned variants into systematic reuse. Neves et al. [87] 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.

## 4 Virtual Platform Overview

Our framework relies on conceptual structures and operators that modify those conceptual structures. 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. 6). In addition, they maintain information about variability, specifically feature

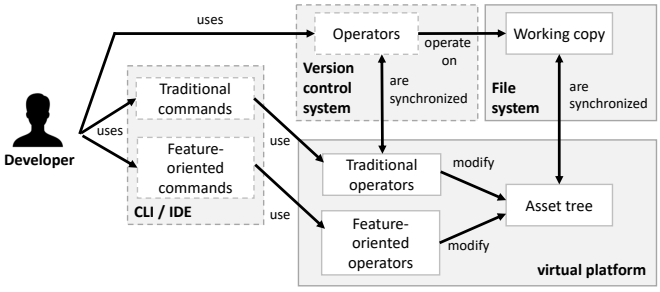



Fig. 3: Overview of developers' interaction with the virtual platform and the framework's inner working (dashed boxes represent optional parts)

information, feature-to-asset mappings, and clone traces. On top of these, 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. 9, we study this trade-off.

The virtual platform extends other development tools, specifically, IDEs and version control systems. Fig. 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 (CLI in Fig. 3), where developers can call feature-oriented operations such as *"Create Feature"* directly.

## 5 Methodology

We followed a design-science-like strategy [88] 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 [89], [90] 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 [66], [67], [68], [69], [70], [71], [72], from which we extracted development activities that should be supported by the virtual platform. We also had a series of 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.

**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.

**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.

**User Study** Having received promising cost and benefit analysis results, we conducted a user study to assess: (i) the correctness and efficiency of performing routine developer tasks, with and without using the virtual platform; and (ii) the virtual platform's usability. As a prerequisite for our user study, we extended the virtual platform to allow developer interaction by implementing a command-line interface on top of our framework. We also integrated virtual platform with the file system so it can work with projects in real-time. The results of our user study are reported in Sec. 10.

## 6 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. Fig. 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. 8).

**Asset Tree ($AT$)** is our main conceptual structure and abstractly represents a hierarchy of assets, such as the folder hierarchy, but also the containment structures within source code 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, [91]), which represent source code files. In our case, we define the $AT$ as a hierarchical, non-cyclic tree 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 code files (e.g., classes, methods, and code blocks). Every node represents an *asset* related to the project, such as a folder, a file (e.g., image, source code 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*. Notably, not every language comprises assets of each *AssetType*; some languages might only require a sub-set of these types.

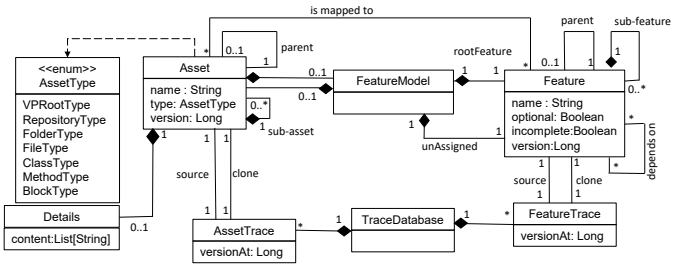



Fig. 4: Conceptual structures: asset tree, features, mappings, and clone traces

The type *VPRootType* is only used once in the $AT$, to specify the synthetic root node. The purpose of this root node is to hold different variants of the software system virtual platform is managing. Virtual platform is designed to use the root when it loads the variants in the $AT$.

Traditional SPLE architectures typically have one feature model per project, which can be difficult to maintain and evolve in large systems (e.g., Linux kernel [22]). 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). This helps prevent having very large feature models at higher levels of granularity (e.g., *RepositoryType* or *FolderType*), while still allowing assets at deeper levels of granularity to have fine-grained features in their respective feature models.

**Well-Formedness Criteria** We define a partial order of valid containment over the types of assets in a check function *containable* : *asset* $\times$ *asset* $\rightarrow \mathbb{B}$ 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. If an operator violates the *well-formedness* criteria, an error message is displayed stating the operator failed to execute as it does not result in a valid containment.

**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. Features, like Assets, are also versioned. A *feature model* ($FM$) 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). Feature models also hold a pointer to their containing asset (see the composition of *asset* in *feature model* in Fig. 4). This is to facilitate developers when locating where a feature is implemented (i.e., in which asset).

**Asset-To-Feature Mappings**, in practice, can have two semantics. They can be simple mapping relationships, indicating that *asset* realizes a *feature* [92]. They can also indicate

variability [93], 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 of mapping assets 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*—a `Long` integer used to recognize changes in the $AT$ (and $FM$), especially among cloned assets. We designed our versioning strategy to align with the file system. Specifically, we retrieve versions from the actual directories and files captured in the $AT$. The version of an asset is the *timestamp* of the last time the asset was modified (retrieved using `file.lastModifiedTime()` in Scala). For simplicity, we track changes at *second* level, implying that *sub-second* level changes are considered to be one super-change occurring in the span of one second. This is however a realistic assumption, as most significant changes take longer that one second (e.g., file renaming, method addition). An illustration of retrieval and pre-processing of last modified times is shown below:

- *get* last modified time → 2022-11-02T11:54:20.709003Z
- *remove* unnecessary information → 2022-11-02 11:54:20
- *remove* format-specific characters → 20221102115420 (asset version)

Assets at a finer level of granularity (e.g., class, method, block) inherit the *version* of the immediate ancestor that has a *version*. Features are versioned slightly differently. For any feature-related change (i.e., invocation of a feature-oriented operator), virtual platform gets the current timestamp from the system, and assigns it to the involved feature(s). The details of versioning for each operator are explained when we present the operators in Sec. 7 below.

This versioning strategy replaces our previous one [52], where we used the *root*'s version to increment and assign versions to modified assets. The initial version of *root* was assigned to be 1 (*version* = 1). With frequent updates occurring in batches, versioning became brittle, so we shifted to a more robust strategy.

**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 the virtual platform, 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.

## 7 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 [51], which also presents additional *convenience operators*—utility methods that efficiently traverse the trees ($AT$ and *feature model*) to return data that needs to be frequently accessed (e.g., assets mapped to a *feature* and clones of an *asset*).

### 7.1 Traditional/Asset-Oriented Operators

We now present our traditional, asset-oriented operators: conventional activities performed by developers during ordinary development. These operators reflect typical developer tasks (e.g., adding a file to a folder), allowing to keep the $AT$ in sync with the working directory. The operators work on the $AT$, which as mentioned above, represents all the assets in the repository in a language-independent tree-like structure. Also, the assets act as *mappable* components to the features, and allow cloning and change propagation. In the following, we introduce the asset-oriented operators with their parameter types, a brief description, and sample scenarios, inspired from our calculator example (cf. Sec. 2.1). The notation used for visualizing various scenarios is shown in Fig. 5.

**`AddAsset`** : *asset* × *asset* → $\mathbb{B}$

*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 assigns the last modified times of the assets represented by $S$ and $T$ (in the file system), to $S$ and $T$ respectively. Notably, it is not always the case that $S$, $T$, or both are file level assets (i.e., *ClassType*, *MethodType*, *BlockType*). In those cases, the assets inherit the version of the closest ancestor asset that has a version. For example if a class *MyClass* is added to a file *MyClass.java*, since we can not use the file system to get the last modified time of the class *MyClass*, the version of the class *MyClass* will be the same as that of the file *MyClass.java*. `AddAsset` also adds any *feature* mapped to $S$ in $T$'s *feature model* (typically repository *feature model*).

*Example:* Recall the *BasicCalculator (BC)* example. The developer adds the implementation for the method *divide* 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 asset *divide* gets the version of the file *Operators.js*. Fig. 6 illustrates this.

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

*Description:* Upon a change in an *asset* $S$ in the repository, `ChangeAsset` retrieves the last modified time of the asset,

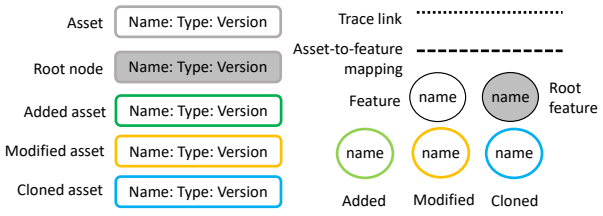



Fig. 5: Notations used in operator illustrations

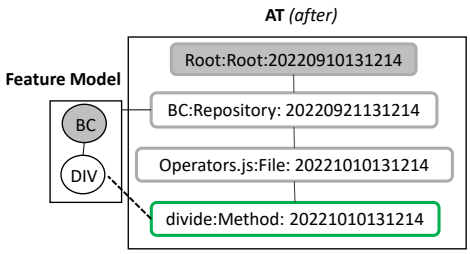



Fig. 6: Illustration of `AddAsset`(*divide*,*Operators.js*)

and assigns it to $S$. Versionable changes include an asset's renaming, addition, removal or modification (e.g., removal of lines of code).

**`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 also gets the last modified time of the parent asset, 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* $S$ (with its sub-assets) to the new target *asset* $T$ (using `CloneAsset`), and removes the *asset* $S$ 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 track of changes through versioning. The following 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. Interestingly, the versioning of this operator is slightly different than that of the previous operators. This is because mappings can be added to an asset in different ways. Features can be embedded internally [94], [92] to the assets, or they can be added externally to the assets [95]. For the latter, the asset itself is not modified, i.e., the last modified time of the asset remains unchanged. So, for `MapAssetToFeature`, we get the current timestamp from the system itself (using `LocalDateTime.now()` in Java), and use it to set the version of the mapped asset. Additionally, the feature also takes the same version (timestamp of when it was added). In fact, for all feature-oriented operators, the versioning of features relies on the local time and date of the system (explained in Sec. 7.2 below).

*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`."

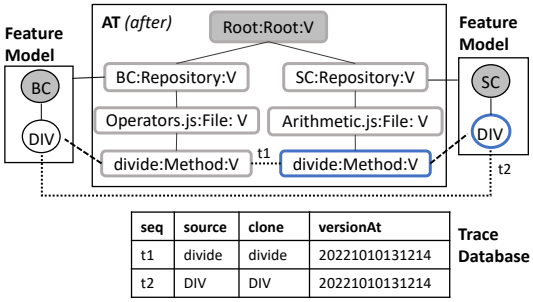



Fig. 7: Illustration of `CloneAsset`(*divide*, *Arithmetic.js*)

**`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), we get the last modified time of the asset from the file system, and assign it to the asset. We also store the trace links by creating traces for both *asset* and *feature* clones and adding 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*. Fig. 7 illustrates the scenario. For brevity, we remove the versions from the illustration, and write `V` instead. The versions of the cloned assets are however depicted in the *TraceDatabase*.

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

*Description:* `PropagateToAsset` takes two assets, checks if one is the 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 version of the target asset is modified and assigned to the asset itself.

*Example:* Assume that the method *divide* method in *BC* during cloning did not include the check for division by

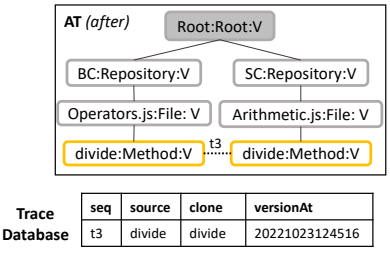



Fig. 8: Illustration of `PropagateToAsset`(*divide*, *divide*)

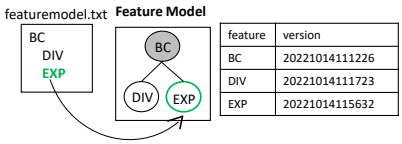



Fig. 9: Illustration of `AddFeature`(*EXP*, *BC*)

zero. After adding the check (`ChangeAsset`), *divide* in the source (*Operators.js*) is ahead (in *version*) of the *divide* in the target (*Arithmetic.js*). By invoking `PropagateToAsset`, the changes are propagated automatically. Fig. 8 demonstrates this; for brevity, *feature* mappings and versions are omitted.

### 7.2 Feature-Oriented Operators

The feature-oriented operators add 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`. The added feature gets the current system time as its version.

*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 [96]). 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 *feature EXP* takes the current timestamp as its version. Fig. 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 model* to $FM$. This is recognized as a change in the $AT$ (`ChangeAsset`), and as such, the asset (to which the feature model is added) gets the current system time as its version.

*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 $FM$ (using `MapAssetToFeature`).

**`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. This is to ensure that if a *feature* is deleted, all assets containing only the implementation of the deleted *feature* are also deleted. However, if the developers want to keep the asset, they can unmap the asset from the feature (using *unmap*, our convenience operator), before deleting the feature. After *feature* removal, `RemoveFeature` assigns the timestamp of removal (system's time) to the parent *feature* of the removed *feature*.

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

*Description:* When a feature is changed, `ChangeFeature` locates the *feature* in the *feature model*, and assigns the current timestamp from the system to the changed *feature*. Notably, these changes are the changes except for addition and removal of sub-features in a *feature*. In `ChangeFeature`, we consider *feature* renaming, and mapping to a new asset (`MapAssetToFeature`).

**`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 them into other projects. `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 clones (of assets and features) retain the same versions from source in the target. However, both the parent *asset* and parent *feature* get the current timestamp of the system as their version.

*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 Fig. 10. For brevity, we replace the versions of the assets with `V`. 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.

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

*Description:* `PropagateToFeature` replicates the changes in the *feature* (e.g., renaming, adding or removing sub-features) to selected or all of its clones. It also propagates new mappings; a new asset mapped to the feature in the source is cloned to the target and mapped to the feature clone. Checking whether the propagation is valid and necessary relies on two conditions based on the *TraceDatabase*: whether one

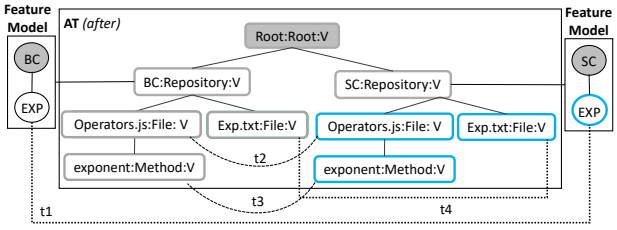



Fig. 10: Illustration of `CloneFeature`(*EXP*, *SC*)

of the features provided is a clone of the other, and whether the *feature* was modified after cloning (current *version* > *versionAt*). Notably, a change in a feature also implies a change in the asset(s) implementing it, and vice versa. Therefore, `PropagateToFeature` invokes `PropagateToAsset` (for all its mapped assets), and vice versa.

After propagating changes, it creates new traces between the source and newly modified targets (both *feature* and *asset*), and adds them to the *TraceDatabase*.

**`DiffAssets`** : $asset \times asset \rightarrow List[asset]$

*Description:* `DiffAssets` takes two assets as input, and returns a list of sub-assets (from $S$ and $T$) that implement *unique* features. In essence, it is an extension of the traditional *git diff*, but aligned to features. Given two assets $S$ and $T$, `DiffAssets` traverses their feature models to find features which are unique to a particular asset. This is particularly useful in the context of variants, when often, developers want to know how close in implementation variants are (e.g., for deciding which variant to clone to create a new one).

## 8 Implementation

Our Scala-based implementation of the virtual platform provides an API as the main interface to execute operators. In the production-ready tool, this API is usable as a command line interface with a set of predefined IDE commands (explained below). We used the language processing programming library kiama (github.com/inkytonik/kiama) [97], [98] 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 a case study of Clafer Web Tools (detailed in Sec. 9 below). We checked correctness by comparing the resultant state (of the $AT$, $FM$, *TraceDatabase*, and mappings) after operator invocations to the expected one. We also simulated the illustrative example presented in Sec. 2.1 by automatically realizing all the discussed scenarios.

Once we implemented and tested our operators, we extended the virtual platform with file-system persistency support and a command-line interface for user interaction. We implemented the persistence over the *AssetTraceDatabase* and enabled virtual platform to write changes back to the file system (a.k.a. *serialization*). We also streamlined our change-propagation strategy to allow cloning and propagation in assets at a finer level of granularity (specifically, code assets, such as classes, methods, and code blocks).

### 8.1 *TraceDatabase* Persistence

We implemented persistence for the TraceDatabase in the form of a text-based file to permanently store and refer to the traces for invoking operators (e.g., `getClones`, `CloneFeature`). We store one trace per line, each comprising the *name* of the source, the clone, and the version of source when it was cloned. We use the same file for storing both asset and feature clones. When storing asset names, we store the path of the asset in the file system, relative to the root asset. Continuing with our calculator example, consider that we clone a file "`Operators.js`" from *BC* to *SC*. Consequently, virtual platform will add a trace to the *TraceDatabase* (which we call "`trace-db`" in the file system).

For feature traces, we only write the feature name, appended to the variant which comprises that feature (with special characters "::"). Extending on the same scenario we presented in the description of `CloneFeature` (where a developer cloned the feature *EXP*), the following trace would be stored in the *TraceDatabase*: `BC::EXP;SC::EXP;20221018123425`

### 8.2 Serialization

We defined an operator `SerializeAssetTree`, which, given an asset, writes the entire $AT$ down to its leaf nodes, also persisting the feature-related information along the way. The input to this operator is typically the root asset. As a first step, `SerializeAssetTree` creates a root folder (if there isn't one already), and then, it proceeds to write the variants. For each asset, it also checks if there is an associated feature model with that asset. If it finds a feature model, it writes the feature model in Clafer syntax [99]. Any new information (e.g., added features), changed information (e.g., feature renaming), or removed information (e.g., removed features) that resulted from invoking any of our feature-oriented operators, which otherwise was not materialized, is written in serialization as well. Specifically, we follow the notation prescribed in [100] when serializing feature-related data. The *TraceDatabase* is also updated, recording traces pertaining to new clones. When serializing code-level assets, we preserve the indentations and line spaces we read in deserialization.

Importantly, `SerializeAssetTree` is not invoked explicitly. It is typically invoked as a silent operator in the background of other operators, such as `CloneFeature` and `PropagateToFeature`. In fact, in our command-line integration, we serialize the $AT$ after every command. While presently, serialization completely overwrites the $AT$, it could be made partial (to only overwrite the modified assets) in order to optimize the performance of the tool.

### 8.3 Code-Level Cloning and Change Propagation

When cloning a feature (between variants), virtual platform also clones all the assets mapped to the feature into the target variant (i.e., using `CloneFeature`). In traditional `CloneAsset`, if we clone a code asset, we provide a target asset. Additionally, if the target asset is a code-asset itself (e.g., block of code in a method, or method in a class), we also provide a *sibling* asset (preexisting in the target) after which we should insert the clone. However, when cloning a feature, `CloneAsset` is invoked in the background, and there, virtual platform needs to smartly decide where to clone the code asset in the target asset. Following, we present how we tackle this issue.

For *blocks of code*, if we are **cloning them to a file** (implying they are not inside any class), we check if they are library imports. If so, we add the block (*block_i*) before the first class in the file. If they are not import statements (e.g., code comments), we either retain their order (from the source) in the target, or, we clone them at the end of the file. The former is possible if the target file is large enough such that the *order of block_i in source <= number of sub-assets in target*. If we are cloning the block **into a class**, we add them to the start of the class. This is to mirror traditional Java code style, where the blocks (e.g., variable names) precede methods in a class. If we are cloning **into a method**, we attempt to find an existing code block in the same method that is mapped to the same feature. If we find one, we clone the block of code after that preexisting block. If there is no existing block of code implementing the same feature, we clone the block of code right at the end of the method.

When cloning *methods* into classes, we try to find a preexisting method that is mapped to the same feature. If we find such a method, we clone the method (we wanted to clone) right after the preexisting method. Otherwise, we clone the method at the end of the class. When cloning *classes* into files, we always add the class at the end of the file.

### 8.4 Command-Line Interface

We implemented a command-line interface for the virtual platform. We relied on the library picocli (picocli.info), which offers a concise way of writing commands with minimal code, in a one-file-per-command format. We implemented every command together with its parameters and defined a short description of what the command does. We first created commands for operators that allowed adding or removing feature-related data. These included `AddFeatureModelToAsset` (command name *addfm*), `AddFeature` (command name *add*), and `RemoveFeature` (command name *remove*). Next, we added the capability of mapping assets to features (command name *map*). Next, imitating the interactivity of *Git*, we defined an *init* command which is used to initialize the virtual platform on a given root. *init* (i.e., *vp init*) command loads the assets into the $AT$, delegating to the right parser based on the programming language used to implement the project. The command *init* also loads the feature models and feature mappings. Lastly, it loads traces from the file `trace-db` into the *TraceDatabase*.

We then defined the *exploratory* commands, which are commands that query the metadata and return the queried information. The command *getmapped* retrieves and displays all assets mapped to a feature, and requires a feature as parameter (e.g., *vp getmapped BC::EXP*). Similarly, *getclones* takes an asset (or a feature), and returns its clones from all variants. The command *gettrace* gets the last trace between a given asset (or feature) and its clone (e.g., *vp gettrace BC/Operators.js SC/Operators.js*). The *gettrace* command also returns the last timestamp at which these clones were synchronized. Lastly, *diff* gets the difference between an asset and its clone, in terms of the unique features they implement (e.g., *vp diff BC SC*).

Lastly, we implemented commands for cloning and propagation. The command *clone* is used for cloning both assets and features e.g., *vp clone SC::EXP BC*), whereas *propagate* is used to automatically propagate changes between clones (e.g., *vp propagate SC::EXP BC::EXP*).

## 9 Simulation Study

We prototyped and evaluated the virtual platform quantitatively using a simulation study based on revision histories from clone & own-based system. Details of our implementation and evaluation are available in the online appendix [51]. We used an open-source system called Clafer Web Tools (CWT, [101]) that was evolved using clone & own in three cloned variants (*ClaferMooVisualizer*, *ClaferConfigurator*, *ClaferIDE*) towards an integrated platform (*ClaferUICommon-Platform*), 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. [92] 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 [51].

### 9.1 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.

### 9.2 Cost & Benefit

As costs, we measure the additional effort imposed on developers by the virtual platform. Our traditional, asset-oriented operators (left-hand column of Table 2) 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 2): 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. We first discuss the observed costs (in RQ1 and RQ2) and benefits (in RQ3 and RQ4) before weighing them off to estimate the total benefit in Section 9.3.

**RQ1:** What are the costs of maintaining features using feature-oriented operators?

TABLE 2: 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 |

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 2, there are 724 invocations of 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 require 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% of all invocations), and 25 relevant mappings for `AddFeature` (12 relevant invocations, 4.0% of all invocations). 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 [92] 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 [102]. Tool-based clone detection requires manual verification and postprocessing, since even the most advanced clone detection tools have imperfect precision and recall [103].

### 9.3 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 approximate based on plausible estimates: (1) For the cost of feature location, we rely on the earlier literature estimate [92] 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 represents most systems of comparable size (i.e., 54.7K lines of text) and number of variants (four). Many product-line migrations of comparable-sized cases are reported in the catalogue in [104]. For larger systems, the representativeness still holds, as our case undergoes all evolution activities typical in most software systems, which are also supported by other frameworks (detailed comparison in [51]). Still, the virtual platform is evaluated much more thoroughly.

### 9.4 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 that simulates 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 [105], [106], [107].

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 [108], [109]). 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.

## 10 User Study

Aiming to evaluate the virtual platform with actual users, we conducted a user study. It comprised an experiment to investigate the correctness and efficiency of the virtual platform for routine developer tasks. Figure 11 shows a high-level overview. Specifically, we conducted a one-factor crossover experiment with 12 participants where the treatment was the mode of work (i.e., *'manual mode of work'* versus *'using VP'*). We used manual mode of work as a baseline for our comparison for several reasons. First, none of the closely related frameworks (cf. Sec. 3) offered support for all the different tasks in our experiment. Second, we aimed to compare our framework against the current practice, which is the 'manual mode of work.' Developers often resort to manual effort for performing tasks such as feature location and clone detection. Third, some activities supported by other frameworks (and virtual platform), such as platform migration (which are also not very common in practice) were deemed too complex and out of the scope for our evaluation due to time constraints. Lastly, it is customary in software engineering tool experiments to use the 'manual mode of work' as a baseline [110], [111], [112], [113], [114]. At the end of the experiment, we asked the participants to assess the virtual platform's usability on a five-point Likert scale and articulate on their choices in free-text format. Finally, following our experiment, we conducted a one-to-one interview session with each participant. We treated the interview responses as additional qualitative data to our qualitative responses (received via our *questionnaire*, detailed under "Supplementary Material").

### 10.1 Experimental Setup

**Research Questions.** Our goal was to compare the correctness and efficiency of performing the tasks manually versus using the virtual platform. We formulated three research questions as follows.

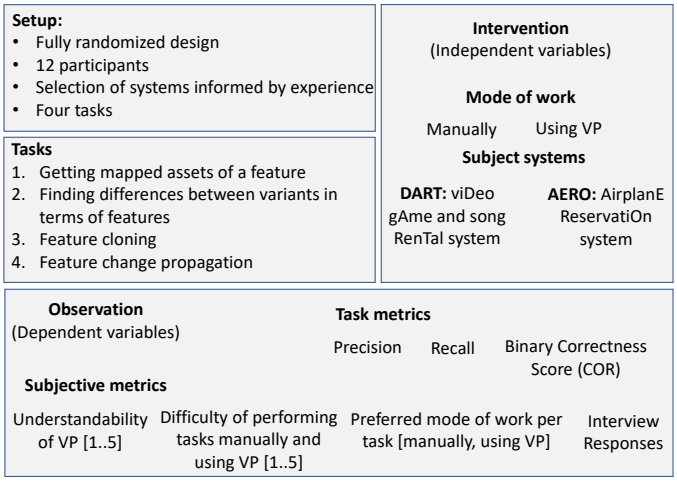



Fig. 11: Overview of our experiment (**VP:** Virtual Platform)

> **RQ5:** How does using virtual platform affect the correctness and efficiency of routine developer tasks compared to the manual mode of work?

We investigated whether using the virtual platform improves task correctness and efficiency while performing routine developer activities as follows: We evaluated the participants' responses and assigned them a numeric score for each task and each mode of work (i.e., metrics values for correctness and efficiency, explained below). We then compared the obtained numeric scores from performing the tasks manually with those from performing them with virtual platform.

Next, we investigated the participants' subjective assessments about the two modes of work with the tasks.

> **RQ6:** How are the modes of work perceived during the various tasks?

We elicited these assessments on a five-point Likert scale. Specifically, we asked about the understandability of virtual platform and the difficulty of performing each task, both manually and using virtual platform.

Next, we asked about preferences regarding the mode of work for each task with the following research question.

> **RQ7:** How are participant preferences for mode of work distributed over different tasks?

We gathered both qualitative and quantitative data about participants' preferences by asking them for their preferred mode of work and the intuition behind choosing it.

**Experimental Design and Implementation.** We opted for a crossover trial [115], which is a variant of the within-subjects design [116], in which all participants receive the treatment (modes of work) at both levels (*manual* and *using virtual platform*). We chose the crossover design since it requires fewer participants than parallel (e.g., between-subject) designs to achieve similar statistical power. The design is less sensitive to variation among participants (e.g., concerning participants' programming expertise). However, a crossover design poses

the risk of *learning effects*; participants can transfer the experience gained from one task to its subsequent tasks. To mitigate such a risk, we randomly allotted half of the participants to first perform the tasks manually and then using the virtual platform, and the other half to perform the tasks in reverse order. Precisely, for two subject systems (*DART* and *AERO*, presented below), and two modes of work (manual and using virtual platform), half of the participants followed *path 1*, and the remaining half followed *path 2* as follows:

- **Path 1:** DART manual → AERO using virtual platform
- **Path 2:** DART using virtual platform → AERO manual

Each participant experimented with each subject system and each work mode only once. Following our one-factor crossover design, we kept the order of subject systems consistent between the groups, to avoid any biases that may arise due to the complexity of the systems. We asked participants to take a 15-minute break between both treatments.

**Participants.** We recruited 12 participants, comprising seven Ph.D. students, three M.Sc. students, two B.Sc. students, and one practitioner. We predominantly employed students as participants due to their suitability as stand-ins for practitioners [117]; students can perform comparable to practitioners for experiments involving unfamiliar software engineering tools. The students were recruited from Chalmers & University of Gothenburg in Sweden, and Ruhr University of Bochum in Germany. All participants had passed programming courses with hands-on experience in Java. No participant had any experience with variant management. We informed each participant before the experiment that participation was entirely voluntary, and that the data would be saved anonymously. Nine participants completed the experiment on a designated computer; three used their own computer.

**Preliminary Assessment.** In the beginning of the questionnaire, we asked participants to self-assess their programming expertise on a five-point Likert scale. The five-point Likert scale comprised positive integers from 1–5 (1 lowest, 5 highest level of expertise). Participants rated their programming expertise as an average of 3.33 (mean) $\pm$ 0.62 (standard deviation).

**Supplementary Material.** Our participants received six supplementary material documents in the form of handouts, which can all be found online in our replication package [51]. First, the tutorial slides (*"VirtualPlatformIntroduction.pptx"*) described the general idea of virtual platform, introduced virtual platform's operators, and showed how to invoke its commands through examples. Additionally, since our participants did not have any experience with variant management, we illustrated the concepts (e.g., change propagation and feature cloning) intuitively using examples to demonstrate the typical tasks in variant management. We also gave our participants a soft copy of the slides and asked them to view the slides in slideshow mode to have the full effect of animations, which we employed to simplify the visualizations. Second, to help participants acclimatize with the subject systems, we provided a handout that comprised details of the subject systems (*"SubjectSystemsIntroduction.pdf"*). Specifically, for each subject system, we presented a feature model and provided each feature's brief description. These feature descriptions were also supplemented with concrete feature locations, down to the file level. The rationale was to make sure that participants' performances were not compromised due to the time required to locate features in code. Both subject systems had three variants each, all of them differing in terms of some unique features. We introduced the variants at a high level and explained each variant's unique features. Third, we created a separate handout (*"OperatorsUsage.pdf"*) that explained the commands, the format to invoke them, and the format for specifying assets and features. As a fourth document, we shared an installation guide (*"VPInstallationGuide.txt"*) with participants who could not participate in-person (on account of them being in another country) and were therefore required to follow it before the experiment.

TABLE 3: Overview of the subject systems

| DART | | | AERO | | |
|---|---|---|---|---|---|
| **Variant** | **Lines** | **Features** | **Variant** | **Lines** | **Features** |
| **DartBasic** | 1332 | 14 | **AEROAlpha** | 666 | 14 |
| **DartPlus** | 1449 | 16 | **AEROBeta** | 728 | 16 |
| **DartPro** | 1512 | 16 | **AEROGamma** | 854 | 17 |
| **Mean** | 1431 | 15 | | 749 | 16 |

All handouts mentioned above were shared with the participants *before* the experiment, both as a soft-copy as well as printouts. Right before starting the experiment, we provided participants with instructions to follow for conducting the experiment. Both groups received different instructions as the order of treatments was reversed between groups. Additionally, participants who conducted the experiment on our designated machine ("InstructionsDesignatedPC.txt") received different instructions than the participants conducting the experiment on their own machines ("InstructionsOwnPC.txt"). Lastly, we implemented our experiment design in the form of an online questionnaire that was provided to the participants right before the experiment. To facilitate the participants and to minimize the effort it takes to switch between screens, we also provided participants with printouts of the questionnaire, so they only need to fill the online questionnaire at the end. The design and flow of the experiment as implemented in the form of an online questionnaire is shared in Fig. 12.

**Subject Systems.** We chose subject systems that were intuitive enough to be understood easily. Both were implemented in Java. Table 3 shows descriptive statistics of them.

**DART** is a desktop application for a rental store, allowing customers to browse items (video games and songs), rent them, and pay for them when returning. DART was implemented by a group of undergraduate students (separate from the students participating in the experiment) at the University of Gothenburg as a term project. It can also used by store managers and store employees. Managers use it to oversee item sales and manage employees. Employees use it for inventory management, customer management, and sales monitoring. DART has three variants: *DARTBasic*, *DARTPlus*, and *DARTPro*. DARTBasic has all the functionalities detailed above. DARTPlus was created by cloning DARTBasic, but it also allows users to message each other (feature *"Messaging"*). Additionally, DARTPlus has enhanced *Security* in that it automatically generates unique passwords for customers rather than prompting the employees to create them. Lastly, DARTPro was also cloned from DARTBasic, but was enhanced to allow employees to import and export

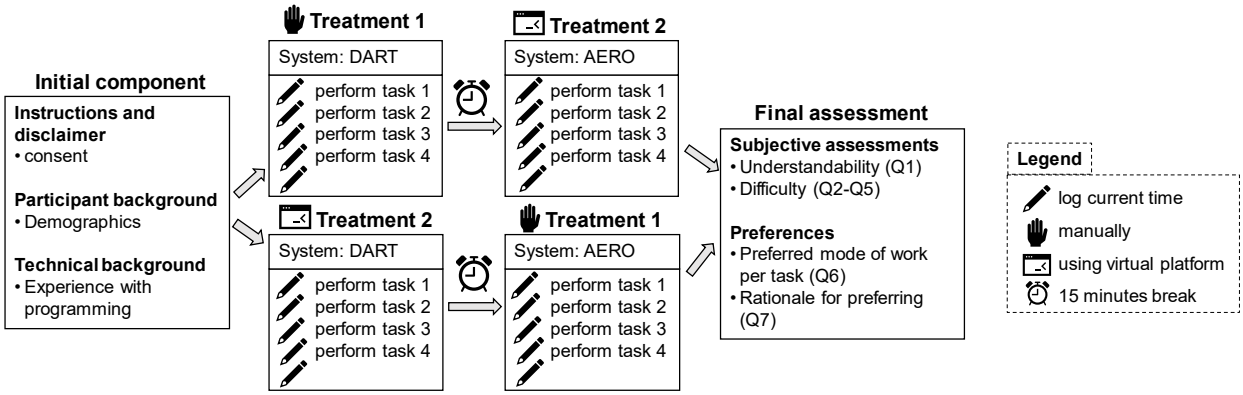



Fig. 12: Design and flow of our experiment

data from and to files, respectively. Notably, the system DART system has more lines of code than the system AERO—understandably because of more stakeholders (i.e., employees, manager, and customers), different classes of customers (i.e., silver, gold, and platinum, skipped due to irrelevance), and elaborate exception handling. However, to minimize the effort it takes to understand and navigate the codebase, as mentioned above (in supplementary material), we provided the locations of features in a handout (*"SubjectSystemsIntroduction.pdf"*). We also deliberately kept the number of features in both subject systems to be comparable.

**AERO**[*] is also a desktop application. It is an open-source project intended to be used by an airline reservation company. Employees use it to manage flights and schedule them on specified days, while customers reserve flights. AERO has three variants: *AEROAlpha*, *AEROBeta*, and *AEROGamma*. AEROAlpha has all the functionalities described above. AEROBeta was created by cloning AEROAlpha, but it also allows employees of the airline reservation company to customize the appearance of their interface. It also has the ability to generate and save tickets to a separate directory. AEROGamma was also created by cloning AEROAlpha, however, it allows users (airline companies, customers) to provide feedback. Additionally, it allows managers to view feedback (comments and ratings). Lastly, it allows employees to load data about various trips and scheduled flights from files.

**Tasks.** We designed our tasks by taking inspiration from the literature and our own experience [92], [118]. We aimed for representativeness with our four tasks. Our first two tasks are *exploratory* in that they do not require making any changes in the code base. The last two tasks are however *hands-on*; participants are required to change the code base when performing those tasks. Below, we discuss our tasks, providing a rationale for each, and giving concrete examples representing each task. For each task, we specified if participants needed to perform it manually or using the virtual platform.

*Task 1 ("Getting mapped assets of a feature")* required participants to traverse the code base of a variant, and find all assets that implement the given feature. This task is an instance of *feature location*, which is a typical and frequent developer task [119]. Feature location is often performed as a lead up to some other task, such as feature enhancement [118]. An example of this kind of task is: *"Find all the assets in AEROBeta that are mapped to the feature Customize."*

[*]https://github.com/CSEMN/AirlineReservationSystem

*Task 2 ("Finding differences between two variants in terms of features")* required participants to find how two variants differed in terms of the unique features. This difference could be new assets added to implement a unique feature (only in one variant), or existing assets modified to implement the unique feature. Developers often need to differentiate variants in terms of features to, for instance, determine which variant to clone to create a new variant. While the traditional *diff* is adequate for returning code-level differences, developers still need to manually dissect those line-oriented diffs to determine whether they correspond to a unique feature, or whether they are simple refactorings (e.g., as done in our prior work [118]). An example of this task kind is: *"How do the variants DartBasic and DartPlus differ in terms of functionality?"* We asked participants to find assets of all kinds, down to code-level, that differ on account of unique features. For both Task 1 and 2, we also specified a response format. For instance, for assets that were not blocks of code, participants could simply write the name of the asset (e.g., *customize()*). For blocks of code, they needed to mention the containing file, and copy and paste the lines of code.

*Task 3 ("Cloning a feature")* required participants to clone a feature (and all assets which implement it) from one variant into another. This feature was different than the feature they found in Task 1. Feature cloning is a typical developer task (feature propagation in [49][†]); developers often clone features that are well-received to other variants. In this task, participants had to traverse the codebase, find all locations of the feature (mapped assets), and copy each asset in the target variant. The goal was to have a running code base with the cloned feature working in the target. An example of this kind of task is: *"Clone the feature Messaging from DARTPlus back into DARTBasic. Please note that Messaging is a scattered feature and as such, assets of different types (classes, methods, blocks) need to be cloned. Your goal is to have an error-free code at the end of the cloning. After finishing, log the ending time in the widget below."*

*Task 4 ("Propagating change in a feature")* required participants to replicate changes in a cloned feature, to the clone of that feature. The changes to the original feature occurred after the feature was cloned, leading to inconsistencies in the feature and its clone requiring synchronization via propagation. Features often undergo evolution [120], e.g., in

[†]In the paper, the authors refer to feature cloning as *feature propagation*, not to be confused with propagation in our paper, which refers to *change propagation* after cloning

TABLE 4: Oracle for Task 1 and Task 2

| Task | | Subject system DART | AERO |
|---|---|---|---|
| **1** | **Feature** | ViewCatalogue | Customize |
| | # of assets | 6 | 5 |
| **2** | **Variants** | DARTBasic, DARTPlus | AEROAlpha AEROGamma |
| | # of assets | 25 | 12 |

the form of maintenance or enhancement. If the changes work well (e.g., after alpha or beta testing), they are propagated to other variants too. In this task, we gave a description of the change, and asked participants to locate the change in the codebase, and propagate it to a specified variant. An example of this task is: *"AEROBeta enhances the feature Book_Flight so the system automatically generates a ticket against each reservation and stores it in a folder "Tickets" (automatically created). As such, this is a change in the feature Book_Flight. Your task is to propagate this change from AEROBeta back into the original variant AEROAlpha using virtual platform. Once you finish the task and you see that the message that the feature was propagated successfully, please log the time in the widget underneath."*

**Task Metrics.** To address RQ5, we elicited several task metrics. Specifically, we used precision, recall, binary correctness score (*COR*), and time taken to measure the correctness and efficiency respectively. For the exploratory tasks (Task 1 and 2), we measured precision and recall, as these tasks required participants to retrieve a number of assets from the codebase (mapped assets for Task 1 and assets implementing unique features for Task 2). Table 4 shows the *oracle* for Task 1 and 2; the total number of mapped assets to the specified features for both subject systems and the number of assets implementing unique features in the specified variants, respectively. Notably, the oracle was established as a result of recurrent discussions between the first and second author, and verified by the other authors. For Task 1, we measured the *correctly identified mapped assets* in proportion to the total number of retrieved mapped assets (precision) and assets implementing the given feature (recall). A response was scored between 0 and 1, and represented the precision and recall ratios. For instance, for Task 1 in DART, if the participant retrieved 3 out of 6 actual mapped assets, their recall was 0.50. For Task 2, we also measured precision and recall. We calculated the number of *correctly identified assets implementing unique features* in proportion to the total number of such retrieved assets (precision) and the actual number of assets implementing unique features (recall). Responses for Task 2 were also scored between 0 and 1, e.g., for Task 2 in AERO, if the participant retrieved 10 assets implementing unique features, 6 of which were correct, their precision was 0.60.

For the hands-on tasks (Task 3 and 4), we calculated a binary correctness score (*COR*). The goal of both tasks was to obtain a running code base with the cloned or propagated feature working in the target variant. We measured correctness on a binary scale: if the feature operated properly in the target variant, the task response got a score of 1, otherwise 0. For both subject systems, the features to clone (*Task 3*) and propagate (*Task 4*) allowed testing by interacting with the system after the task was performed. For instance, for *DART*, participants were required to clone the feature *"Messaging"*, allowing users to send messages to each other. We validated if the feature worked properly by checking that a) the feature was present in the menu options, and b) the feature worked as intended upon interaction with it (e.g., sending a message and reading it for the feature *"Messaging"*). Our rationale for using a binary score was as follows: even subtle implementation mistakes can lead to extensive extra effort (e.g., extensive debugging or static analysis) to be detected and fixed. Consequently, an almost-correct solution could be as problematic as a fully incorrect one. Notably, for Task 3, we do not measure completeness, and only measure the correctness (on a binary scale as explained above). This is because in our supplementary materials for subject systems' introduction (*"SubjectSystemsIntroduction.pdf"*), we point to the locations of the files containing the feature implementations, enabling the participants to find the locations of the features easily. With known feature locations, there is little possibility that participants would miss any assets mapped to the feature. Similarly, for Task 4, we deliberately kept the task easy, requiring participants to simply replace the implementation of a method with the updated implementation from the original variant. Since there was only one method, we did not use the completeness metric for this task as well, only opting for the correctness score.

Finally, to measure task efficiency, participants were asked to log task start and end times in the questionnaire.

**Subjective Metrics.** For RQ6, we asked participants about the *understandability* of the virtual platform and the difficulty of performing each task both manually and using the virtual platform. Specifically:

***(S1)*** How easy did you understand the virtual platform?
***(S2)*** How difficult was it to perform Task 1 (getting mapped assets of a feature) manually and using virtual platform?
***(S3)*** How difficult was it to perform Task 2 (variant diffing) manually and using virtual platform?
***(S4)*** How difficult was it to perform Task 3 (feature cloning) manually and using virtual platform?
***(S5)*** How difficult was it to perform Task 4 (change propagation) manually and using virtual platform?

Following the common convention used for subjective assessment, we elicited the responses on a five-point Likert scale. We used consistent labels over the Likert scales in the questions, stating explicitly for each question that 1 means *"very easy"* and 5 means *"very difficult"*. The questions were also followed by text-fields asking participants to elaborate on their choices. Notably, we did not have two Likert scales for **S1**, as we expected each participant to fully understand what manual mode of work means. For **S2-S5**, we put two Likert scales per question, one for each mode of work. With our subjective assessments regarding difficulty (*S2-S5*), we sought to investigate the following hypothesis: *while performing all four tasks, participants experienced same amount of difficulty with both modes of work*. To this end, we conducted a statistical analysis to compare the difficulty of performing each task manually and using virtual platform.

For RQ7, we asked participants to specify their preferred mode of work for each task. To gain a deeper insight into the intuition behind their choice, and to augment the quantitative data with qualitative information, we also asked to elaborate on their rationale. We formulated the following questions:

***(S6)*** How do you prefer to perform each of the tasks?
***(S7)*** Can you explain your subjective preferences intuitively?

For **S6**, we used two literals (manually, using virtual platform) for each task. We kept **S7** open-ended (with long-paragraph format) to allow participants to write descriptively. We share an analysis over the qualitative responses we received for **S1**–**S7** in Sec. 10.4.4.

Lastly, as mentioned, to gather more insights about the experiment, we conducted one-to-one interviews with each participant. The interviews took place right after the participants submitted the questionnaire. Participation was voluntary: we asked each participant if they would like to have a brief interview regarding their experience in the experiment. We also asked if they needed a break before participating in the interview. In all cases, the participant chose to have the interview right away. The interview was semi-structured to allow participants to think more freely and consequently enable us to retrieve richer and more diverse data. Each interview took roughly 10 minutes. The interview was led by the primary author, who took notes during the interview, also rephrasing participants' responses back to them to ensure that she understood their responses. Specifically, the interview featured a combination of the following questions, posed in no specific order:

***(I1)*** What mode of work did you prefer and why?
***(I2)*** What command does the virtual platform shine the most in your opinion? What command does it fall short in?
***(I3)*** How difficult is the virtual platform to learn?
***(I4)*** If virtual platform was taken to the industry, would it be accepted?
***(I5)*** Are there any suggestions for improvement?

We treated the participant responses to **I1**–**I5** as qualitative feedback, the analysis of which is shared in Sec. 10.4.5.

**Pilot Study.** Prior to conducting the user study, we performed a pilot study with one participant. The goal was to recognize potential problems with the design of the experiment and its materials, including the questionnaire design and the interaction with the tool. In line with what we entailed above, we shared the supplementary materials with the participant before the experiment. Moreover, we shared the subject systems and the questionnaire with the participant right before he started the experiment. Once the participant carried out the experiment, we conducted a one-to-one interview with him to gain insights about his experience.

Based on participants' responses in questionnaire and follow-up interview, we resolved three issues. First, after the participant mentioned that the output of commands included redundant information, we significantly trimmed down the output of the relevant commands. Second, we fixed a `NullPointerException` when trying to access the *TraceDatabase*. Third, the the formulation of some tasks was misleading in that it was difficult to understand which mode of work the participants were required to use to perform it. So we made the mode of work more explicit in the descriptions.

### 10.2 Analysis

Following the convention for statistical analysis, we initiated our analysis by establishing the normality of our distributions. To this end, we employed the Shapiro-Wilk test for normality [121] for our dependent variables that were numeric (precision and recall for Task 1 and 2, and time taken for all tasks). Notably, Task 3 and 4 were evaluated on a binary scale (*COR*), and hence, did not require establishing normality. Notably, the precision, recall, and *COR* of all tasks using virtual platform was always 1 (the maximum), also verified after a post-experiment analysis of the variants on which virtual platform was used. For the results with the manual mode of work, we observed that the distribution of precision of Task 1 was homogeneous (*p-value* = 0.31) and that of Task 2 was skewed (*p-value* < 0.001). Additionally, both distributions for recall (Task 1 and 2) were homogeneous (Task 1: *p-value* = 0.68, Task 2: *p-value* = 0.23). Despite observing some homogeneous distributions, due to the skewness in the responses using virtual platform, we used non-parametric tests to determine if there was a significant difference between the dependent variables using both modes of work.

For hypothesis testing for Task 1 and 2 (*RQ5*), we used Wilcoxon rank sum test [122], a standard non-parametric test for comparing two related samples. We used the standard significance threshold of 0.05. For quantifying the difference (i.e., effect size), we employed the $A_{12}$ measure, which is a standard non-parametric effect size measure suggested by best practices for statistical testing [123], [124] when all samples are not normally distributed, which was the case in our experiment. We followed the Vargha and Delaney's original three interpretations [125]: $A_{12} \approx 0.56 = small$; $A_{12} \approx 0.64 = medium$; and $A_{12} \approx 0.71 = large$. For hypothesis testing for Task 3 and 4 (*RQ5*), we employed Fisher's Exact Test [126], since the *COR* values are categorical data, taking values of 0 and 1 for incorrect and correct responses respectively. We used the $A_{12}$ measure for quantifying the magnitude of difference in responses for Task 3 and 4.

For comparing the time taken using each mode of work (*RQ5*), we first checked the normality of the distributions. We observed that for Task 1 (manual: *p-value* = 0.032, using virtual platform: *p-value* = 0.003) and Task 2 (manual: *p-value* = 0.92, using virtual platform: *p-value* = 0.002), either or both of the distributions were skewed. In contrast, time distributions for Task 3 and 4 using both modes of work were homogeneous (*Task 3:* 0.32 vs 0.06, *Task 4:* 0.08 vs 0.07). Consequently, we used the Wilcoxon rank sum test to compare the samples for Task 1 and 2, and the parametric t-test [127] for comparing the samples for Task 3 and 4. We also employed the $A_{12}$ measure for quantifying the difference.

For the subjective assessments (*RQ6*), as previously mentioned, we retrieved two kinds of data: the difficulty ratings and the qualitative responses. For the former, we conducted normality tests for each task using the Shapiro-Wilk test. For Task 1 and 2, the distributions for the samples using manual mode of work were homogeneous (*Task 1*: 0.12, *Task 2*: 0.05). The sample distributions of Task 1 and 2 using the virtual platform were both skewed (*Task 1* < 0.001, *Task 2* < 0.001). For Task 3 and 4, the distributions of samples using manual mode of work were also homogeneous (*Task 3*: 0.06, *Task 4*: 0.06), however, the test could not work on the distributions of Task 3 and 4 using virtual platform, because all values were identical (very skewed). We opted for the Wilcoxon rank sum test for comparing the difficulty ratings of all tasks, and the $A_{12}$ measure for quantifying the degree of difference between the ratings.

For the qualitative assessments, one of the authors performed *inductive coding* to tag the participant's comments with keywords. Once the process was completed, the other

TABLE 5: Precision (**Prec**) and recall (**Rec**) of Tasks 1 and 2 manually and using virtual platform ($T1_{inc}$: number of incorrectly identified mapped assets in Task 1, $T2_{inc}$: number of incorrectly identified assets mapping unique features in Task 2)

| | **Manually** | | | | | | | | **Using virtual platform** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Task 1** | | | **Task 2** | | | **Task 3** | **Task 4** | **Task 1** | | **Task 1** | | **Task 3** | **Task 4** |
| Participants | Prec | Rec | $T1_{inc}$ | Prec | Rec | $T2_{inc}$ | $COR_{m3}$ | $COR_{m4}$ | Prec | Rec | Prec | Rec | $COR_{v3}$ | $COR_{v4}$ |
| Participant 1 | 0.29 | 0.83 | 12 | 0.95 | 0.80 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Participant 2 | 0.80 | 0.80 | 1 | 0.88 | 0.67 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Participant 3 | 1.00 | 1.00 | 0 | 0.93 | 0.56 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Participant 4 | 0.25 | 0.40 | 6 | 0.92 | 1.00 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Participant 5 | 0.71 | 0.83 | 2 | 0.43 | 0.28 | 9 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Participant 6 | 0.50 | 0.20 | 1 | 1.00 | 0.58 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| Participant 7 | 0.75 | 0.50 | 1 | 0.86 | 1.00 | 4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Participant 8 | 1.00 | 0.60 | 0 | 1.00 | 0.92 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Participant 9 | 0.50 | 0.17 | 1 | 0.95 | 0.76 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Participant 10 | 0.60 | 0.60 | 2 | 1.00 | 0.58 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Participant 11 | 1.00 | 0.50 | 0 | 0.75 | 0.13 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Participant 12 | 0.75 | 0.60 | 1 | 1.00 | 0.08 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| **Mean** | 0.67 | 0.58 | 2.25 | 0.89 | 0.61 | 1.58 | 0.66 | 0.83 | 1 | 1 | 1 | 1 | 1 | 1 |
| **Std dev.** | 0.24 | 0.25 | 3.30 | 0.15 | 0.30 | 2.46 | 0.50 | 0.40 | 0 | 0 | 0 | 0 | 1 | 1 |

authors verified if the keywords accurately captured the gist of the responses. If there was a disagreement, the authors were able to reach a consensus via discussion. The tags assisted in identifying the pertinent aspects, and their frequency helped determine the redundant concerns. We used R for running our quantitative analysis. The analysis scripts can be found in our replication package [51].

### 10.3 Ethical Considerations

We carefully discussed ethical considerations in the team of authors. Importantly, our study does not have problematic ethical implications for the involved participants, as we did not expose our study participants to threats to their physical or psychological well-being, did not process protected personal data, and explicitly asked them for their consent to participate in the study. Formally, we addressed the relevant local regulations at the institutions where the study was performed and where it was analysed. The study was performed in universities in Sweden and Germany. In Sweden, the relevant regulation is the Ethical Review Act (Etikprövningslagen EPL, SFS 2003:460), whose paragraph 4 describes the criteria under which research performed in Sweden requires an ethical review by a review board. Our research does not fall under these criteria. In particular, we did not perform any physical or psychological interventions that lead to potential harms to humans—our target of investigation was participant behavior in relation to software-engineering activities. Germany does not have a comparable national-level regulation, except for specific fields such as medical devices and drugs, which are not related to our study. Institutional regulations and the corresponding review boards are at the level of individual faculties, most commonly in medicine. At the faculty where the study was performed, no relevant regulation exists. Beyond the execution of the experiment, authors from other institutions were involved with analysis of the produced data; however, they worked with fully anonymized versions of the data.

### 10.4 Results

We now present the results of our user study.

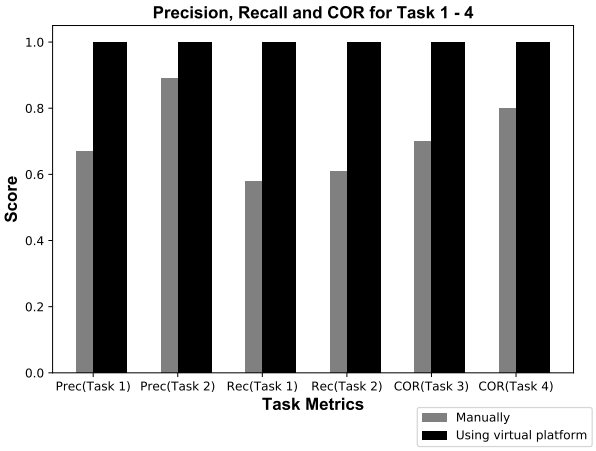



Fig. 13: Task metrics for Task 1 - 4 (**Prec:** precision **Rec:** Recall)

#### 10.4.1 RQ5: Correctness and Efficiency

For RQ5, we measured precision, recall, *COR*, and time taken for each task using both modes of work. Precision and recall are marked on a scale from 0 to 1, *COR* is a binary value (0 or 1), and time is measured in minutes. Table 5 provides a high-level overview of the results we discuss in the following.

**Precision and Recall.** Figure 13 shows the average precision and recall scores for Task 1 and 2.

In **Task 1** (getting mapped assets of a feature), participants performed better with the virtual platform (mean precision and recall = 1.0) than manually (mean precision = 0.67, mean recall = 0.58). The difference is statistically significant for precision (*p-value* < 0.001) and recall (*p-value* = 0.003), with a high-ranged effect size for both ($A_{12}$ = 0.87 and 0.95, respectively). This implies that developer tasks that rely on feature location (e.g., feature enhancement, feature cloning) are highly prone to inaccuracies if the feature is identified manually.

Moreover, we observed that participants also retrieved false positives (assets that were not mapped to the given feature, but were perceived to be), depicted by $T1_{inc}$ in Table 5. This resulted in lower precision values, which implies that developers can apply unwarranted changes to assets not mapped to a feature, incorrectly assuming they are mapped to that feature, during routine activities, e.g., feature evolution. A low recall indicates that relying on manual feature location can lead to partially complete

results, for instance, partially cloning a feature or changing only a partial set of assets mapped to the feature.

In **Task 2** (getting differences between variants in terms of the unique features), participants also performed better using virtual platform (mean precision and recall = 1.0) than using the manual mode of work (mean precision = 0.89, mean recall = 0.61). The difference in precision (*p-value* < 0.001) and recall (*p-value* < 0.001) was statistically significant, and also the effect size were high for both ($A_{12}$ = 0.83 and 0.92 respectively). This implies that manually differentiating variants, e.g., for determining which variant to clone for creating a new one, can lead to inaccuracies. Additionally, we noticed that participants retrieved some false positives (assets that were common between variants, but participants thought they were not), demonstrated by $T2_{inc}$ in Table 5. This resulted in lower precision values, indicating that for routine developer tasks, e.g., performing a commonality and variability analysis for extractive adoption [47], developers can potentially integrate features that are not shared into one (merged) feature. A low recall implies that developers can miss integrating some features, thinking they are not shared between variants when in fact they are (false negatives, i.e., missed assets).

> ***Finding 1***. *Virtual platform outperformed the manual mode of work for both exploratory tasks, obtaining statistically significant results. In addition to missing some correct responses, participants also retrieved some incorrect ones, indicating that the manual mode of work lacks reliability, and might introduce inaccuracies and inconsistencies for activities that follow exploratory tasks e.g., feature cloning.*

**Binary Correctness Score (COR).** Figure 13 (right bars) represents the average values of *COR* for Task 3 and 4. Considering **Task 3** (cloning a feature), participants on average performed better using the virtual platform than performing the task manually (mean *COR* = 1.00 vs 0.66). The difference was however not statistically significant (*p-value* = 0.09), with a medium-ranged effect size ($A_{12}$ = 0.66). The medium-ranged effect size is possibly because participants were required to have the code running, and the feature operational, and they could use the compiler to determine if they had cloned the feature correctly. Still, one-thirds of our participants could not clone the feature, and stopped after running into errors. Specifically, the participants either failed to clone the new import statements (e.g., for the feature *"Messaging,"* all the classes representing users, i.e., different types of customers needed to import the class "`Message.java`,") or they failed to update the existing methods (e.g., for the feature *"Messaging,"* the class constructors also changed to require setting an empty inbox when the customer was created). This implies that copying a feature from one variant to another manually can be error-prone, and lead to inconsistencies in the code base (e.g., partial cloning of a feature). While we chose the size and complexity of the subject systems to align with the scope of our experiment, we believe that the inconsistencies associated with feature cloning will be escalated for larger, more complex systems.

For **Task 4** (propagating changes in a feature), participants performed better on average, albeit unnoticeably, using the virtual platform than propagating the changes manually (mean *COR* = 1.00 vs 0.83). The difference was not statistically significant (*p-value* = 0.47), with a small-ranged effect size ($A_{12}$ = 0.58). This could be because participants gained significant familiarity with the subject system after performing the preceding tasks. Additionally, the feature locations provided in the supplementary material bypassed the effort and time required to locate features manually in code. Lastly, to aim for simplicity, we designed the task at method-level instead of line-level, so participants only needed to copy paste the entire content of the method (implementing the change in the feature) in the target variant. Still, two participants could not locate the change manually, or stopped after running into errors. We believe that the challenges would be amplified for assets at finer levels of granularity (e.g., blocks and lines of code), and complex systems with many scattered features.

> ***Finding 2***. *Virtual platform outperformed for both hands-on tasks. The differences were not statistically significant. However, multiple participants ceased to perform both feature cloning and feature change propagation on account of errors (e.g., package dependencies, order of methods) after unsuccessfully trying to solve them manually.*

**Time.** Table 6 presents an overview of the completion times of participants when performing all four tasks of our experiment. For all tasks, participants were faster when using the virtual platform than performing the tasks manually. The total time of the tasks using the manual mode of work ($T_{manual}$) was also always greater than the time taken by the same participant using the virtual platform ($T_{VP}$). Figure 14 shows a visual representation of the comparison between times taken using both modes of work for all four tasks.

In **Task 1**, participants performed notably faster on average using the virtual platform than with the manual mode of work (mean = 4.91 vs 9.41). The difference was also statistically significant (*p-value* = 0.03), with a large-ranged effect size ($A_{12}$ = 0.75).

In **Task 2**, the differences were far more pronounced. On average, participants performed remarkably better using the virtual platform than finding the feature-oriented differences manually (mean = 4.75 vs. 18.41). The difference was also statistically significant (*p-value* = 0.002), with a high-ranged effect size ($A_{12}$ = 0.93). These findings also align with the correctness scores above, where participants performed worse using the manual mode of work. This is still surprising, considering we provided the information of unique features and (at an abstract level,) where they were implemented. We speculate that the differences would be even more prominent for larger, more complex systems.

In **Task 3**, we noticed a similar pattern; participants performed notably faster when using the virtual platform than cloning the feature manually (mean = 14.58 vs 2.66). The difference was also statistically significant (*p-value* = 0.002), with a high-ranged effect size ($A_{12}$ = 0.98). The average time with the manual mode of work also validates previous research [92], where the average time taken to manually locate one feature was estimated to be 15 minutes.

In **Task 4**, participants also took less time on average using the virtual platform, albeit not very prominently, than with the manual mode of work (mean = 3.66 vs 5.58).

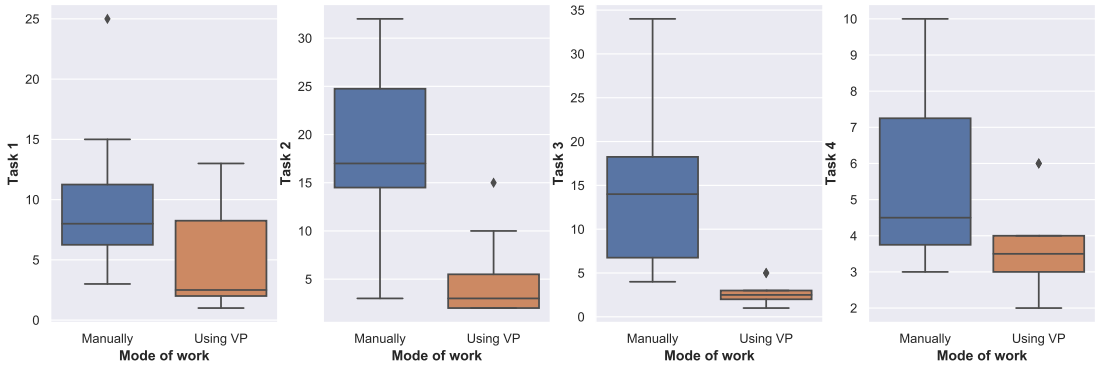



Fig. 14: Times taken for Task 1 - 4 manually and using the virtual platform (**VP:** virtual platform)

TABLE 6: Time taken (in minutes) for tasks 1, 2, 3, and 4, manually and using virtual platform ($T_{manual}$: Total time using manual mode of work, $T_{VP}$: Total time using the virtual platform, TT: Total time of the experiment)

| | **Manually** | | | | | **Using virtual platform** | | | | | **Total** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Participants | Task 1 | Task 2 | Task 3 | Task 4 | $T_{manual}$ | Task 1 | Task 2 | Task 3 | Task 4 | $T_{VP}$ | TT |
| Participant 1 | 10 | 17 | 14 | 7 | 48 | 13 | 15 | 2 | 4 | 38 | 86 |
| Participant 2 | 7 | 17 | 7 | 5 | 36 | 11 | 4 | 2 | 4 | 25 | 61 |
| Participant 3 | 25 | 24 | 34 | 4 | 87 | 9 | 5 | 3 | 6 | 23 | 110 |
| Participant 4 | 4 | 28 | 6 | 8 | 46 | 4 | 10 | 5 | 3 | 22 | 68 |
| Participant 5 | 8 | 27 | 14 | 3 | 52 | 3 | 2 | 2 | 3 | 10 | 62 |
| Participant 6 | 8 | 21 | 22 | 9 | 60 | 8 | 3 | 2 | 3 | 16 | 76 |
| Participant 7 | 15 | 32 | 16 | 3 | 66 | 2 | 3 | 1 | 4 | 10 | 76 |
| Participant 8 | 3 | 10 | 4 | 4 | 21 | 2 | 2 | 1 | 2 | 7 | 28 |
| Participant 9 | 15 | 10 | 23 | 4 | 52 | 2 | 2 | 5 | 6 | 15 | 67 |
| Participant 10 | 8 | 16 | 6 | 7 | 37 | 1 | 7 | 3 | 4 | 15 | 52 |
| Participant 11 | 7 | 16 | 17 | 3 | 43 | 2 | 2 | 3 | 2 | 9 | 52 |
| Participant 12 | 3 | 3 | 12 | 10 | 28 | 2 | 2 | 3 | 3 | 10 | 38 |
| **Mean** | 9.41 | 18.41 | 14.58 | 5.58 | 48 | 4.91 | 4.75 | 2.66 | 3.66 | 16.66 | 64.66 |
| **Std dev.** | 6.02 | 8.07 | 8.34 | 2.39 | 16.94 | 2.39 | 5.25 | 3.89 | 1.24 | 8.58 | 20.77 |

Hypothesis testing did not reveal any significance in the difference (*p-value* = 0.06), however, the difference was still large-ranged ($A_{12}$ = 0.73). Participants being relatively faster in this task could be explained by the learning effect; they were more acclimatized to the subject system by the time they performed this task. We noticed a similar pattern with virtual platform; the average times taken for the latter two tasks are less than those for the first two tasks.

> ***Finding 3***. *Participants were faster in performing their tasks using virtual platform on average than when using the manual mode of work with statistically significant results for Task 1, Task 2 and Task 3. Effect sizes for all tasks were high-ranged.*

#### 10.4.2 RQ6: Subjective Perception

Regarding understandability, participants rated the virtual platform to have a mean score of 1.83 (lower scores indicating better understandability) on the Likert scale. 10 participants gave a rating less than or equal to 2, whereas two participants gave a rating of 4. However, when explaining their reasoning behind the provided rating, both participants wrote positive reviews: *"The commands fit well to the description. Though terminology in the field might differ, e.g. propagate is maybe called merge. Steps [were] clear which feature to work on."* We therefore speculate that those participants misunderstood the scale, thinking a higher rating reflected better understandability (despite labels being collocated with the ratings to indicate what they meant). We now present the results for the subjective ratings regarding difficulty.

Table 7 presents a summary of our difficulty ratings for performing all four tasks using both modes of work. The ratings were gathered in response to the assessment questions (S2-S5) in the questionnaire. The responses were gathered on a five-point Likert scale, with lower scores indicating less difficulty. Fig. 15 shows a comparison of average difficulty ratings using both modes of work for different tasks.

Considering **Task 1**, the manual mode of work was considered more difficult than virtual platform on average (mean = 1.5 vs 3.3). The difference was also statistically significant (*p-value* = 0.01), with a high-ranged effect size ($A_{12}$ = 0.90). For **Task 2**, we noticed the same trend occurring more notably. Participants perceived manual mode of work to be much more difficult than using virtual platform (mean = 3.7 vs 1.5). The difference was also statistically significant (*p-value* = 0.002), with a high-ranged effect size ($A_{12}$ = 0.97). For **Task 3**, we observed a similar pattern. Participants found that cloning the feature manually was much more difficult than cloning it using virtual platform (mean = 3.2 vs 1.0). The difference was also statistically significant (*p-value* = 0.002), with a very high-ranged effect size ($A_{12}$ = 1.0). For **Task 4**, participants again found the manual mode of work to be notably more difficult than using virtual platform for propagating changes in a feature (mean = 2.5 vs 1.0). Hypothesis testing revealed significance in the difference (*p-value* = 0.005), with a high-ranged effect size ($A_{12}$ = 0.91). Interestingly, all participants selected the minimum rating

(1.00: *very easy*) for both Task 3 and Task 4. The results align well with the correctness scores and the times taken in *RQ5*.

> ***Finding 4***. *The participants perceived the manual mode to be more difficult on average for performing all tasks. The differences were statistically significant for all tasks, with high-ranged effect sizes. For Task 3 (feature cloning) and Task 4 (feature change propagation), all participants chose the minimum difficulty rating (1) for the virtual platform.*

#### 10.4.3 RQ7: Subjective Preferences

We report on the distribution of participant preferences over both modes of work for the four tasks based on **S6** in our questionnaire. In S6, we asked participants to state their preferred mode of work for every task (quantitative data). Following, we present an analysis of the distribution of participants' preferences per task.

Table 8 shows a summary of participant preferences for our tasks. Evidently, virtual platform was preferred over the manual mode of work for all four tasks. The preferences were all increasingly skewed, with all participants preferring virtual platform over the manual mode of work for the hand-on tasks (Task 3 and Task 4). This indicates trust; developers when performing *invasive* tasks can get wary of the potential side effects their changes could have on the system. With more participants favoring virtual platform over the manual mode of work, it is evident that they consider it to be a more reliable solution. At least 80 percent of the participants preferred virtual platform over the manual mode of work for each task. Intuitively, the observed proportions are not surprising. Virtual platform, as we showed in Sec. 10.4.1, is a more efficient and effective solution. It is also easier to use, as observed in Sec. 10.4.2.

> ***Finding 5***. *Participants preferred virtual platform over the manual mode of work for majority of the cases for all four tasks. The distribution was at least 83.3% participants for each task. In line with the difficulty ratings, all participants preferred virtual platform over the manual mode of work for Task 3 and Task 4.*

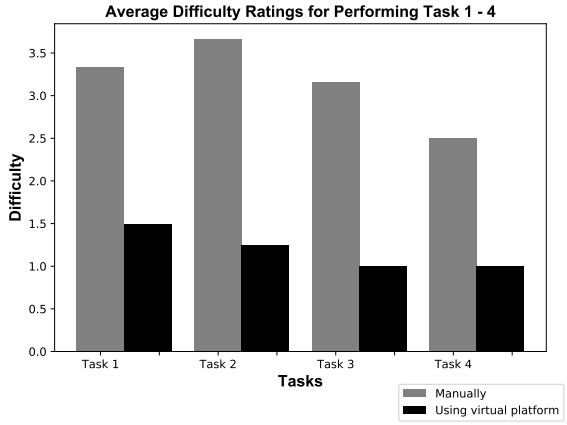



Fig. 15: Difficulty ratings (Task 1–4)

#### 10.4.4 Rationale for Subjective Preferences

In **S7**, we asked participants to provide their intuition behind preferring one mode of work over the other. We conducted a manual analysis of each participant comment, and extracted the key concerns referred to in it. Subsequently, we gathered aspects that were recurrently deemed important by our participants. Following, we present our analysis over the qualitative data. Our findings can help guide developers on when to use virtual platform over the manual mode of work, what to expect from the tool, and what kind of interaction it provides.

**Conciseness.** Participants found virtual platform's commands to be concise, remarking that they only needed to write a single line to perform a particular task: *"Performing the tasks with VP only requires a line of code, compared against the manual approach which requires deeper study and navigation of the code."* However, some participants found the output of the virtual platform to be cluttered: *"For the exploration the virtual platform took me a while to understand the structure of the output, but once that is completed it gets a lot easier compared to searching all relevant files for single lines of code that could be relevant."*

**Ease.** Participants remarked that it was easy to use virtual platform: *"Once [the concepts of cloning, propagation and mapping] are mastered, the syntax of the virtual platform commands [is] really easy to understand and use."* On the contrary, participants found the manual mode of work to be intensive: *"The command line tool was just me tying a command and looking at the results. On the other hand manually looking into the files was quite cumbersome."* Additionally, participants also mentioned experiencing difficulty using the manual mode of work: *"Manually, [variant diffing] was extremely difficult, as the assets were much more scattered through multiple classes. Using the tool made it very clear what the differences are."*

**Understandability.** Participants deemed virtual platform to be easy to understand as well: *"The commands are easily understandable, given the documentation."* They also remarked that the commands aligned well with the tool's intent: *"The commands are only a handful, self explanatory for easy understanding, and map very well with the tool's purpose."* The concepts are also deemed simple by the participants: *"The operators in the virtual platform are quite simple to understand, the introduction slides and the document containing the operators help understanding it better."* In contrast, manual mode of work was pronounced to be a complex solution: *"Performing the*

TABLE 7: Difficulty ratings for Task 1-4, manually and using the virtual platform (Std dev.: Standard deviation)

| Task | Manually | | Using virtual platform | |
|---|---|---|---|---|
| | Mean | Std dev. | Mean | Std dev. |
| Task 1 | 3.3 | 1.1 | 1.5 | 1.2 |
| Task 2 | 3.7 | 1.0 | 1.2 | 0.5 |
| Task 3 | 3.2 | 1.0 | 1.0 | 0 |
| Task 4 | 2.5 | 1.1 | 1.0 | 0 |

TABLE 8: Distribution of participant preferences over the two modes of work for performing various tasks

| Task | Manually | Using virtual platform |
|---|---|---|
| Task 1 | 16.67% | 83.33% |
| Task 2 | 8.33% | 91.66% |
| Task 3 | 0% | 100% |
| Task 4 | 0% | 100% |

*tasks with VP only requires a line of code, compared against the manual approach which requires deeper study and navigation of the code, building the source code to ensure nothing breaks, and more. All the required manual steps highlights the complexity of manual approach over VP."* Additionally, we observed that the supplementary material was well-received: *"Going through the concrete examples in the slide helped me understand the concepts of assets, and features."* Most participants found the terminology in the supplementary material and command-line interface intuitive, however, some of them found the term "propagate" to be ambiguous: *"In general the commands were intuitive. Although propagate could be a term that [is not] used by practitioner."*

**Time.** Many participants favored the virtual platform over the manual mode of work owing to its efficiency: *"Cloning involved a lot of copy-pasting, which is error-prone. Even though I think I did everything correctly, it was a time-consuming task."* In contrast, participants favored virtual platform because it was a more efficient solution, as demonstrated in Sec. 10.4.1: *"I would say that I prefer the VP since it takes less time, manually doing these tasks also is very fragile and error prone."* One participant remarked that the slow execution time of the virtual platform was a hindrance in the experiment: *"One small annoyance is the long execution time for each command."* For most experiments however, this was not reported as a concern, so we assume that the slow execution time could be because of factors such as memory overload by another heavy program, or multiple programs in parallel. Trial runs on our PC (16.0 GB memory, Core i5-8350U CPU, 64-bit operating system, x64-based processor) show that the command execution times for all tasks were under three seconds (mean time for: *Task 1* = 2.1 seconds, *Task 2* = 2.2 seconds, *Task 3* = 2.9 seconds, and *Task 4* = 2.5 seconds), the first response coming in the first second of execution. These times comply well with the thresholds defined by Miller [128], who said: *"Although the user should be informed by the system within four seconds that it has understood and can interpret the command, its execution and final confirmation to the user that the command has been executed may have long and variable delays of minutes."* Still, investigating into ways in which virtual platform can be made even faster could be fruitful.

**Error-proneness.** Participants also mentioned that the manual mode of work was prone to inaccuracies and inconsistencies: *"It is much easier to get something missed if we don't have an automatic tool like [virtual platform]"*. In line with this, participants commented that virtual platform can help avoid inaccuracies and inconsistencies: *"I also think that using the virtual platform is less susceptible to errors than performing the tasks manually."*

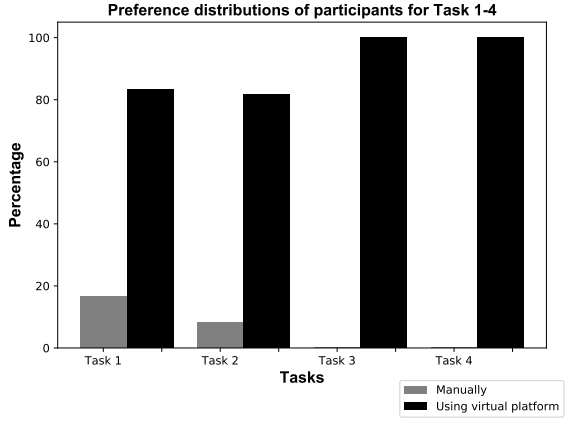



Fig. 16: Participant preferences (Task 1 - 4)

**Certainty.** Many participants reported feeling uncertain about the completeness of their task responses when using the manual mode of work: *"Propagating features manually is very hard. One is always unsure whether something is missing. This doesn't happen with virtual platform."* Contrarily, participants demonstrated trust when using the virtual platform: *"You do not have to worry that you forgot something anywhere and you can be sure that a feature will be cloned/propagated in its entirety."* Interestingly, two participants also highlighted the issue of mistrust associated with the virtual platform, especially in the context of exploratory tasks: *"VP makes [each task a] black-box operation. The outcome is therefore less predictable and requires trust in the correct mapping of the feature. Testing and potentially code review is required in both cases."*

**Usability.** Participants found the command-line tool to exhibit good usability: *"Subcommands are labeled conveniently (based on my perception based on common naming schemes). Furthermore, dynamic parameters are clearly labeled—Version::Feature is uniquely identifiable. Overall, the platform makes an "easy to use and configure" appearance."*

**Task-specific.** We observed that participant preferences also vary between tasks and contexts. For instance, a participant favored the manual mode of work over virtual platform when propagating changes. They also mentioned that this could possibly be because the changed feature to be propagated was implemented in an isolated method: *"Manually, I found it pretty easy to propagate the feature, since the feature was not scattered along multiple class files. [...] it could be way harder if it was scattered even more. The virtual platform made it a bit easier, as it was way faster."*

**Other aspects.** Some participants preferred the virtual platform because of the level-of-detail of the output: *"Using the tool [makes it] very clear what the differences are [in variant diffing]. It also provides the actual location of the differences."* One participant remarked that having knowledge of the codebase made it easier to clone features, albeit slowly: *"If you know what differences you need to copy over it is not a problem even manually, but it takes more time."* Participants also speculated that virtual platform will be more scalable when performing the tasks in larger, more complex systems: *"Manually, [feature propagation] was not too difficult, but I think this is partly because there wasn't much code to copy over. I can imagine that an even more scattered feature will take much more time and also is more error prone."* Extending on this, many participants reported that the scatteredness of features was a hindrance when performing tasks such as feature cloning: *"Again, it is much easier to using a single command to clone compared to manually coping methods, and ensuring all the reference to copied symbols are also cloned. The manual approach is prone to error and requires more time to ensure completeness."* Moreover, confirming our speculation, participants found the error highlighting provided by the compiler to be helpful: *"[...] during the manual exploration I overlooked some relevant parts of the code. The error highlighting in IntelliJ was helpful to guide me to the missing parts."* We also received comments suggesting that implementing command assistance over the command-line interface could be useful: *"For some of the vp commands it was*

*not immediately recognizable if you have made a typing [mistake], especially when specifying the parameters."*

> ***Finding 6***. *Participants reported a multitude of reasons for favoring virtual platform over the manual mode, including the conciseness of input and ease of use. They remarked that the tool was easy to understand with the documentation, and it exhibited more time efficiency. Contrary to this, they remarked that the manual mode was error-prone, leading to uncertainty. Other factors behind the participants' choices included usability, familiarity with the codebase, scalability, scatteredness, and the nature of the task.*

#### 10.4.5 Interview Responses

As previously mentioned, to attain validation and additional insights about the user study, we triangulated with each participant by conducting one-to-one interviews. We now discuss our analysis of the responses participants provided to the interview questions **I1-I4**. We provide our analysis on the potential improvements (**I5**) in Sec. 10.5.

**General preference of mode of work and its rationale.** We asked participants (in **I1**) about their general preferred mode of work, which was the virtual platform in all cases, as observed in the subjective preferences above (Sec. 10.4.3). The reasoning behind participant choices was multi-fold. In addition to the factors mentioned above (Sec. 10.4.4), participants mentioned that one factor for preferring virtual platform was its uniqueness: *"I know [from my own experience] that there are very few tools for feature location and also very less focus on that area. I think VP would be really useful to the industry."* One participant liked the fact that the modality of virtual platform resembled that of Git: *"[I like it because it] feels like Git, [the] style of [specifying] commands is like Git."* Most participants remarked that the commands of virtual platform, including how parameters are specified, was easy to grasp: *"[I was] pretty happy especially with wording of [the] commands. A lot of softwares try to establish their own terminology which is sometimes very counter-intuitive and does not make much sense. [However, virtual platform] commands were very easy to comprehend."* Participants also reaffirmed that time was a factor in determining their preferred mode of work: *"Aside from the time it took to setup initially, [I] was pretty happy with the speed and accuracy of the tasks."* Other factors that participants mentioned were detrimental in their choice were ease, comprehensibility of documentation, understandability, conciseness, simplicity, and command-relevance.

**Virtual platform: strong and weak suits.** Aligned with the subjective assessments and qualitative responses, most participants considered both hands-on tasks to be the strongest suits of the virtual platform: *"virtual platform would be better [than manual work] especially for cloning and propagation."* The most frequently mentioned task that virtual platform excelled at was feature change propagation: *"All four [tasks] are not far apart [in terms of how good is virtual platform in handling them]. Propagate was the feature I liked the most. If I would do it myself especially for a lot of changes, it [would] probably not [be] easy."* Two participants mentioned that feature-oriented variant diffing is the task that virtual platform falls short in. Subsequently, we investigated participants about the reason for their judgment. One participant mentioned that the output was un-structured: *"The output was a bit cluttered. Adding more structure would help. I could still see the changes."* The other participant remarked that he did not understand the usefulness of the command. Upon explaining the potential use cases of feature-oriented variant diffing (e.g., variability analysis for extractive adoption and finding the most suitable variant to clone to create a newly requested one), the participant agreed that it was a quite useful command, also remarking that it was interesting.

**Learnability.** Participants generally felt that virtual platform was easy to learn. One participant also factored the fact that they were unfamiliar with command-line mode of interaction: *"The command-line interface was very easy [to use] considering I never used command-line interface before."* Participants also remarked that virtual platform would be easier to learn if developers had adequate knowledge of concepts pertaining to software product-line engineering: *" [The concept was] pretty easy [to understand], if you know feature models, you will have no problems understanding [the] virtual platform."* One participant also commented on the estimated time it would take for a developer to learn to use virtual platform using a command-line interface: *"I think it would take half a day to one [full] day to learn the virtual platform."* Interestingly, two participants remarked that as they got acclimatized to the command-line interface, interacting with the virtual platform became easier: *"Specifying parameters [assets and features in the commands] in the beginning is difficult. After some time, I felt comfortable quickly"* and *"In the beginning the commands were difficult. Once I understood them, I got a hold of them."*

**Acceptance.** In general, participants believed that virtual platform would be easily accepted in the industry. Adding their intuitions, one participant remarked: *"[Even] if the company does not have a software product-line yet, they can still use the virtual platform, [which in my opinion is] one of the biggest pro and reason to have it. [...] I think everyone needs to learn about the virtual platform."* Another participant remarked: *"virtual platform would be accepted easily, [as I know from my experience that] developers [in fact] face these problems in real life."* One participant observed that having an existing code base would make it challenging to incorporate the virtual platform: *"If you have an existing code base, then it might be difficult [to incorporate the virtual platform]. Then you would have to take your time to identify features."* We provide guidelines on how developers can integrate virtual platform in an existing codebase at the end of Sec. 10.5. Lastly, one participant observed: *"[The] virtual platform would in general be accepted but may be [an] IDE plugin is [received] better than [the] CLI."*

### 10.5 Discussion and Recommendations

Our subjective assessments and qualitative assessments pave way for recommendations that developers can follow when performing routine feature evolution and variant management activities. They also hint towards potential improvements that can be applied to the virtual platform. Following, we provide a set of guidelines when choosing a mode of work for the above-mentioned activities (which are represented by our tasks). We also present a set of possible improvements.

**For getting a quick overview, use virtual platform.** Based on the quantitative analysis and qualitative feedback, we believe that the information from virtual platform can be an

effective starting point when acclimatizing to a variant-rich system. For instance, virtual platform could be employed to attain a horizontal overview (e.g., features implemented in a variant) as well as a vertical overview (e.g., getting the differences between variants). Especially, owing to its efficiency, we recommend developers to use virtual platform when time is a concern, e.g, quickly finding a feature for post-deployment maintenance or quickly cloning a feature across variants for a release.

**For large systems with scattered feature implementations, use virtual platform.** Virtual platform stores and exploits recorded metadata, bypassing the need to traverse code and locate features. For larger, complex systems, with many features scattered across the codebase, the manual mode of work will not scale up. In those cases, it is advisable to use the virtual platform. However, if the system is small, and developers know the system well, manual mode of work can perform equally good, albeit slower. For the hands-on tasks, if the feature is not scattered across multiple files, and developers have prior knowledge of where the feature is implemented, they can use the manual mode of work (e.g., for cloning a feature or propagating changes in a feature).

**Use virtual platform for risky scenarios.** Often times, developers are dealing with time-sensitive, safety-critical systems (e.g., medical devices, aircraft control systems). In those cases, using manual mode of work can increase the risk (of breaking the code at compile-time or run-time) due to its error-proneness. In those cases, virtual platform can be relied upon as a dependable system.

**Plan activities to build trust in the virtual platform.** As inception, it is crucial that developers develop a sense of trust with the virtual platform. To this end, it is recommended to organize training sessions where developers interact with the virtual platform by using it on simpler cases, e.g., a simpler variant or a slice of the system. Additionally, based on the feedback, the output of the virtual platform can be further streamlined to show only the relevant information, with user-friendly outputs such as success messages.

In our follow-up interviews, we realized that some participants were unsure about the usefulness of a command, especially when they were unaware of the contexts in which it could be useful. Upon explanation, they seemed to agree on the relevance of the command, even commenting that it was an important one. We therefore believe that it is important that during training, developers are made aware of the problems which virtual platform solves, and the contexts where it is applicable. Additionally, it is vital that developers acquire foundational knowledge of concepts specific to software product-line engineering, such as feature models and feature-oriented software development and evolution. These concepts are however simple and relatively easy to learn, as commented by our participants. Lastly, it is important that developers traverse through the documentation for virtual platform (e.g., user manual, tool instructions etc.), as we observed in Sec. 10.4.4 and Sec. 10.4.5 that documentation played a significant role in the comprehensibility of the virtual platform.

**Potential improvements.** Based on participants' feedback (from the questionnaire and in response to **I5**), there are a multitude of potential improvements that can be applied on top of the virtual platform. Generally, they fall under two categories: suggestions that require minor fixes in the current implementation to enhance usability, and suggestions that require major implementation for providing other modes of interaction in addition to the command-line interface.

Among the former, most are related to changing command names to make them more intuitive and structuring the format of the output in a more organized manner. Regarding the command names, the commands *getmapped* and *propagate* were deemed confusing. Consequently, we will change the command names to *findfeature* and *merge* respectively. For the output format, we will further trim down the output, and add more context to reduce the time and effort it takes to comprehend the output of certain commands.

Among the latter, we can augment the virtual platform with IDE support, eliminating the need for developers to switch between tools when working on a system. The IDE support should, in addition to the conventional developer tasks (e.g., adding a file, removing a method), allow developers to perform different feature-oriented tasks (e.g., adding a feature model, adding a mapping between a feature and an asset that implements it). The IDE should also provide support for invoking commands for performing different exploratory as well as hands-on tasks. With a command-line interface and IDE integration in place, a dedicated graphical user interface could be redundant.

Finally, it would be important to guide developers on how to approach virtual platform integration if the organization already has a few variants in place. In that scenario, virtual platform would not harm any existing knowledge base that the organization has, or affect the current state of the development. Companies would be able to reap benefits of the virtual platform as soon as it is integrated. Still, it could be possible, in some cases, to accurately recover metadata pertaining to clone traceability, e.g., by examining the forking history of the system's variants in a version control system. Additionally, once developers start adding features, the feature-to-asset mappings could be used to train virtual platform to suggest feature mappings for unmarked feature code [129], [130].

### 10.6 Threats to Validity

**Construct validity.** We followed a multi-stage design and refinement process to mitigate threats that could arise from the supplementary materials. Firstly, we designed the materials after one-on-one discussions between the first and second authors. Two of the other authors reviewed the materials. We later conducted a pilot study with the fifth author, who did not participate in the development of the virtual platform or the preparation or initial review of the materials. Finally, based on the feedback we got at the end of the pilot study, we refined the materials before launching the actual experiment.

We selected subject systems that are intuitive, moderately complex, and realistic enough to mirror the interactivity of real-world applications. While studying the impact of using virtual platform on larger systems is worth investigating, we believe that it would be out of the scope of a user study due to limitations regarding the cognitive load participants should be exposed to (before experiencing fatigue).

Our task selection was informed by our own experience and the knowledge of frequent developer activities when dealing with variant-rich systems reported in the

literature [92]. Since we observed statistical differences in the performance using both modes of work, we argue that the difficulty of our tasks is fitting; however, other tasks might exist, e.g., deriving an entire variant or cloning an entire feature model. Generally, a consolidated catalog of different variant management and product-line migration tasks would be helpful, but currently, research lacks such a classification.

Lastly, our instrument was a command-line interface—a common mechanism for interacting with tools [131]—also available for many other applications, such as Git.

**Internal Validity.** To mitigate learning effects, we used counterbalancing: we changed the order of treatments (modes of work) between groups while keeping the order of subject systems and tasks. We also randomized the assignment of participants to ensure balance in the groups. Another threat to our internal validity is maturation due to participant fatigue or loss of interest [132]. As a mitigation, we asked participants to take a 15-minute break between exposure to both treatments. Additionally, since three of our participants experimented on their personal computers, we cannot be certain that they experimented in the same setup (e.g., noise level). However, we argue that developers in real life also have to operate in various setups and environments.

We also collected qualitative data to gain insights into, and triangulate the experimental findings. To eliminate the effect of participant fatigue on the interview responses (as the interview was right after the experiment), we set the duration of each interview to 10 minutes, which is considered short. However, the questionnaire at the end of the experiment also contained open-ended questions. Therefore, we could supplement data collected through interviews with participants' responses to the open-ended questions in the questionnaire, to obtain qualitative data sufficient for triangulation.

We chose subject systems that were intuitive and comprised comparable complexity, having an almost identical number of features. Additionally, hypothesis testing (using the Wilcoxon rank-sum test [122]) did not reveal any significant differences for manual mode of work between the performance of participants working on AERO and the performance of those working on DART (in terms of precision and recall). For the virtual platform, the performance of the participants working on AERO, and the performance of those working on DART, did not show a statistically significant difference either. We also applied linear mixed-effects models [133] to investigate whether the interaction of mode of work (VP vs. manual) and subject system (AERO vs. DART) affected the participants' performance. The results did not indicate any statistically significant impact of the interaction on the Task 1 precision (*p-value* = 0.709) and recall (*p-value*=0.484), nor on the Task 2 precision (*p-value* = 0.069) and recall (*p-value* = 0.792). We checked the normality of errors with the Shapiro-Wilk test and visually inspected the residuals' histograms and normal QQ plots for the linear mixed-effects models we built. We tried transformations, including square root, arcsine, logarithmic, and the extension of box-cox transformation to random effect models [134]. However, the data distributions did not satisfy the assumptions that need to hold for the linear mixed effects models. Therefore, although some studies have shown that linear mixed-effects models are robust in the violation of distributional assumptions [135], our findings need interpretation with caution.

**External Validity.** We believe our findings are most representative of developers with similar levels of expertise ($3.33 \pm 0.62$). As we previously argued (in Sec. 10.1), students can act as stand-ins for developers in experiments involving a development approach (such as virtual platform) that is new to both students and professionals [117]. While it would be interesting to consider a more diversified range of experience, we arguably recruit participants from a critical population: Consider that an organization hires developers with programming expertise comparable to our participants. Therefore, our findings can shed light on how a poor understanding of the system or the tasks' nature can affect developer productivity and pose a risk to the organization.

Our tasks mimic frequent developer activities as explained in Sec. 2 and Sec. 10.1. While studying the impact of using virtual platform on a broader category of tasks is left for future work, our results presented in Sec. 10.4 yield a promising perspective for generalizability.

Finally, a potential threat is the generalizability of our findings to larger systems. Since the complexity of the system increases with an increasing number (and scattering degree) of features, we argue that the differences between the correctness and time using virtual platform would be far more pronounced compared to the manual mode of work, also speculated by our participants (Sec. 10.4.4).

**Statistical Conclusion Validity.** Since we conducted our experiment with 12 participants, which is relatively a few, we used a crossover design to gather representative data points to argue for statistically valid results. While selecting the statistical tests to compare two modes of work, we checked if the data complied with the tests' assumptions. The Shapiro-Wilk test indicated that the distribution of correctness values for Task 1 and 2 (i.e., precision and recall) and the efficiency values for all tasks are mostly not normal. Moreover, subjective preferences are nominal data. We used the Wilcoxon rank sum test to compare the precision, recall, efficiency, and subjective preferences for two work modes. We preferred the Wilcoxon rank sum test over Mann-Whitney U Test as conducting our experiment with 12 participants resulted in a small sample size. Finally, we used Fisher's Exact Test to compare the *COR* values of Task 3 and 4, since the data did not comply with the assumptions of the Chi-Squared Test due to the small sample size.

## 11 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. Our user study conducted with 12 participants also leads to promising results in terms of the correctness, efficiency, and usability of the tool. Specifically, using virtual platform for routine developer tasks such as locating/cloning features, distinguishing variants, and propagating changes across variants leads to faster and better results (in terms of precision, recall, and correctness). The tool is generally well-received by the participants, with majority of the participants preferring the virtual platform over the manual mode of work. We provide guidelines on how to incorporate and use virtual platform in Sec. 10.5.

We see several promising directions for 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 [87], which could be extended to support systems in our intermediate governance levels. Specifying our operators in the framework of software product line transformations [136], [137], [138] would make them amenable to conflict and dependency analysis [139], 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 [100], [140], we could facilitate inspection of the available feature mappings. By integrating the virtual platform with view-based and feature-oriented asset-integration techniques, we could enhance the integration of cloned assets by letting developers preview and explore different possible integrations [141]. 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. We plan to also enhance the evaluation of the virtual platform by conducting a complementary expert evaluation using practitioners with significant experience in variability management and product-line migration. Other directions are to support configuration of variants by selecting features, offering views [142], [143], and providing visualizations (e.g., feature dashboards [94], [144], [145]). Notably, our operator-based evolution support gives rise to intelligent recommender systems, which could in the future recommend operators or features to developers, to foster the use of feature-orientation in development, for instance, building upon our recent feature traceability recommender system [146], [147]. While the current implementation of the virtual platform caters for textual assets, it could also be worthwhile to extend virtual platform's capabilities to software design models [148], which are typically used for understanding the system, and even for automating activities such as development and testing. Finally, recommender systems that learn from the meta-data and support developers handling features and assets could further encourage using features in software engineering [147]—or even foster further automation of our operators with the help of large language models [149].

## ACKNOWLEDGMENTS

Supported by Swedish Research Council (257822902), Vinnova Sweden (2016-02804), and Wallenberg Academy. We thank Christoph Derks and Khaled Al-Mustafa for contributing to the framework and command-line interface.

## 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. Businge, O. Moses, S. Nadi, and T. Berger, "Reuse and maintenance practices among divergent forks in three software ecosystems," *Empirical Software Engineering*, vol. 27, no. 2, p. 54, 2022.

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

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

[7] 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.

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

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

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

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

[12] 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.

[13] A. Wąsowski and T. Berger, *Domain-Specific Languages: Effective Modeling, Automation, and Reuse*. Springer Nature, 2023, ch. Software Product Lines, pp. 395–435.

[14] L. Wozniak and P. Clements, "How automotive engineering is taking product line engineering to the extreme," in *Proceedings of the 19th International Conference on Software Product Line*, 2015, pp. 327–336.

[15] R. Flores, C. Krueger, and P. Clements, "Mega-scale product line engineering at general motors," in *Proceedings of the 16th International Software Product Line Conference-Volume 1*, 2012, pp. 259–268.

[16] D. C. Sharp, "Reducing avionics software cost through component based product line development," in *17th DASC. AIAA/IEEE/SAE. Digital Avionics Systems Conference. Proceedings (Cat. No. 98CH36267)*, 1998.

[17] G. Chastek, P. Donohoe, J. D. McGregor, and D. Muthig, "Engineering a production method for a software product line," in *SPLC*, 2011.

[18] F. Dordowsky and W. Hipp, "Adopting software product line principles to manage software variants in a complex avionics system," in *SPLC*, 2009.

[19] S. García, D. Strüber, D. Brugali, A. Di Fava, P. Schillinger, P. Pelliccione, and T. Berger, "Variability modeling of service robots: Experiences and challenges," in *VaMoS*, 2019, pp. 8:1–6.

[20] S. Garcia, D. Strueber, D. Brugali, A. D. Fava, P. Pelliccione, and T. Berger, "Software variability in service robotics," *Empirical Software Engineering*, vol. 28, no. 1, p. 24, 2023.

[21] 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.

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

[23] 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.

[24] P. Franz, T. Berger, I. Fayaz, S. Nadi, and E. Groshev, "Configfix: Interactive configuration conflict resolution for the linux kernel," in *ICSE/SEIP*, 2021.

[25] I. Abal, C. Brabrand, and A. Wasowski, "42 variability bugs in the linux kernel: a qualitative analysis," in *ASE*, 2014.

[26] 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.

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

[28] 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.

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

[30] 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.

[31] 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.

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

[33] 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.

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

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

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

[37] 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.

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

[39] 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.

[40] 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.

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

[42] 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.

[43] 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.

[44] 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.

[45] 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.

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

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

[48] 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.

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

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

[51] "Virtual platform's online appendix," https://bitbucket.org/easelab/2023-vponlineappendix.

[52] W. Mahmood, D. Strüber, T. Berger, R. Lämmel, and M. Mukelabai, "Seamless variability management with the virtual platform," in *ICSE*. IEEE, 2021, pp. 1658–1670.

[53] "The top programming languages," https://octoverse.github.com/2022/top-programming-languages.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

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

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

[62] 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.

[63] 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.

[64] P. M. Bittner, A. Schultheiß, T. Thüm, T. Kehrer, J. M. Young, and L. Linsbauer, "Feature trace recording," in *FSE*, 2021.

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

[66] J. Rubin, K. Czarnecki, and M. Chechik, "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.

[67] 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.

[68] 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.

[69] T. Kehrer, T. Thüm, A. Schultheiß, and P. M. Bittner, "Bridging the gap between clone-and-own and software product lines," in *ICSE-NIER*. IEEE, 2021.

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

[71] C. König, K. Rosiak, L. Linsbauer, and I. Schaefer, "Synchronizing software variants: a two-dimensional approach," in *SPLC*, 2022.

[72] T. Schmorleiz and R. Lämmel, "Similarity management of'cloned and owned'variants," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 2016, pp. 1466–1471.

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

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

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

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

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

[78] 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.

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

[80] 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.

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

[82] 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.

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

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

[85] D. Rabiser, P. Grünbacher, H. Prähofer, and F. Angerer, "A prototype-based approach for managing clones in clone-and-own product lines," in *SPLC*, 2016.

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

[87] 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.

[88] P. Johannesson and E. Perjons, *An introduction to design science*. Springer, 2014, vol. 10.

[89] L. Neves, L. Teixeira, D. Sena, V. Alves, U. Kulesza, and P. Borba, "Investigating the safe evolution of software product lines," in *GPCE*, 2011.

[90] G. C. S. Ferreira, F. N. Gaia, E. Figueiredo, and M. de Almeida Maia, "On the use of feature-oriented programming for evolving software product lines—a comparative study," *Science of Computer programming*, vol. 93, pp. 65–85, 2014.

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

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

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

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

[95] M. P. Robillard and G. C. Murphy, "Representing concerns in source code," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 16, no. 1, pp. 3–es, 2007.

[96] 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.

[97] A. Wąsowski and T. Berger, *Domain-Specific Languages: Effective Modeling, Automation, and Reuse*. Springer Nature, 2023, ch. Model and Program Transformation, pp. 233–292.

[98] A. M. Sloane, "Lightweight language processing in kiama," in *GTTSE*, 2009.

[99] "Clafer documentation," https://www.clafer.org/p/documentation.html.

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

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

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

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

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

[105] 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.

[106] 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.

[107] C. Derks, D. Strüber, and T. Berger, "A benchmark generator framework for evolving variant-rich software," *Journal of Systems and Software*, vol. 203, p. 111736, 2023.

[108] 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.

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

[110] G. Da Silva, S. Correa, A. Da Cunha, L. DIAS, and O. Saotome, "A comparison between automated generated code tools using model based development," in *IEEE/AIAA 30th Digital Avionics Systems Conference*, pp. 7E4–1.

[111] M. Grechanik, Q. Xie, and C. Fu, "Experimental assessment of manual versus tool-based maintenance of gui-directed test scripts," in *2009 ieee international conference on software maintenance*. IEEE, 2009, pp. 9–18.

[112] C. R. L. Neto, I. do Carmo Machado, V. C. Garcia, and E. S. de Almeida, "Analyzing the effectiveness of a system testing tool for software product line engineering (s)." in *SEKE*, 2013, pp. 584–588.

[113] M. Raza and J. P. Faria, "Processpair: A tool for automated performance analysis and improvement recommendation in software development," in *ASE*, 2016.

[114] L. Karlsson, T. Thelin, B. Regnell, P. Berander, and C. Wohlin, "Pair-wise comparisons versus planning game partitioning—experiments on requirements prioritisation techniques," *Empirical Software Engineering*, vol. 12, pp. 3–33, 2007.

[115] D. Raghavarao and L. Padgett, *Repeated measurements and cross-over designs*. John Wiley & Sons, 2014.

[116] B. Jones and M. G. Kenward, *Design and analysis of cross-over trials*. Chapman and Hall/CRC, 1989.

[117] I. Salman, A. T. Misirli, and N. Juristo, "Are students representatives of professionals in software engineering experiments?" in *ICSE*, 2015.

[118] W. Mahmood, M. Chagama, T. Berger, and R. Hebig, "Causes of merge conflicts: A case study of elasticsearch," in *VaMoS*, 2020.

[119] J. Krüger, T. Berger, and T. Leich, *Software Engineering for Variability Intensive Systems*, 2019, ch. Features and How to Find Them: A Survey of Manual Feature Location, pp. 153–172.

[120] D. Hinterreiter, H. Prähofer, L. Linsbauer, P. Grünbacher, F. Reisinger, and A. Egyed, "Feature-oriented evolution of automation software systems in industrial software ecosystems," in *ETFA*, 2018.

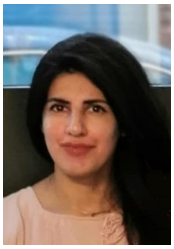
[121] P. Royston, "Approximating the shapiro-wilk w-test for non-normality," *Statistics and computing*, vol. 2, no. 3, pp. 117–119, 1992.

[122] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics*. Springer, 1992, pp. 196–202.

[123] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

[124] M. Shepperd and S. MacDonell, "Evaluating prediction systems in software project estimation," *Information and Software Technology*, vol. 54, no. 8, pp. 820–827, 2012.

[125] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.
[126] R. Fisher, "Statistical methods for research workers. 4th edn edinburgh: Oliver and boyd." 1932.
[127] Student, "The probable error of a mean," *Biometrika*, pp. 1–25, 1908.
[128] R. B. Miller, "Response time in man-computer conversational transactions," in *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, 1968, pp. 267–277.
[129] C. S. Corley, K. Damevski, and N. A. Kraft, "Exploring the use of deep learning for feature location," in *ICSME*, 2015.
[130] A. C. Marcén, J. Font, Ó. Pastor, and C. Cetina, "Towards feature location in models through a learning to rank approach," in *SPLC*, 2017.
[131] M. Schröder and J. Cito, "An empirical investigation of command-line customization," *Empirical Software Engineering*, vol. 27, no. 2, pp. 1–26, 2022.
[132] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.
[133] A. Gałecki, T. Burzykowski, A. Gałecki, and T. Burzykowski, *Linear mixed-effects model*. Springer, 2013.
[134] A. Almohaimeed and J. Einbeck, "boxcoxmix: Box–Cox-type transformations for linear and logistic models with random effects," Tech. Rep., 2020.
[135] S. Garcia, D. Strueber, D. Brugali, A. D. Fava, P. Pelliccione, and T. Berger, "Robustness of linear mixed-effects models to violations of distributional assumptions," *Methods in Ecology and Evolution*, vol. 11, no. 9, 2020.
[136] G. Taentzer, R. Salay, D. Strüber, and M. Chechik, "Transformations of software product lines: A generalizing framework based on category theory," in *MODELS*, 2017.
[137] D. Strüber, S. Peldszus, and J. Jürjens, "Taming multi-variability of software product line transformations," in *FASE*, 2018.
[138] M. Chechik, M. Famelis, R. Salay, and D. Strüber, "Perspectives of model transformation reuse," in *IFM*, 2016.
[139] 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.
[140] J. Martinson, H. Jansson, M. Mukelabai, T. Berger, A. Bergel, and T. Ho-Quang, "Hans: Ide-based editing support for embedded feature annotations," in *25th ACM International Systems and Software Product Line Conference (SPLC), Tools Track*, 2021.
[141] M. Lillack, Ştefan Stănciulescu, W. Hedman, T. Berger, and A. Wąsowski, "Intention-based integration of software variants," in *ICSE*, 2019.
[142] S. Stanciulescu, T. Berger, E. Walkingshaw, and A. Wąsowski, "Concepts, operations, and feasibility of a projection-based variation control system," in *ICSME*, 2016.
[143] B. Behringer, J. Palz, and T. Berger, "Peopl: Projectional editing of product lines," in *ICSE*, 2017.
[144] S. Entekhabi, A. Solback, J.-P. Steghöfer, and T. Berger, "Visualization of feature locations with the tool featuredashboard," in *SPLC, Tools Track*, 2019.
[145] A. Bergel, R. Ghzouli, T. Berger, and M. R. V. Chaudron, "Feature-vista: Interactive feature visualization," in *25th ACM International Systems and Software Product Line Conference (SPLC)*, 2021.
[146] M. Mukelabai, K. Hermann, T. Berger, and J.-P. Steghoefer, "Featracer: Locating features through assisted traceability," *IEEE Transactions on Software Engineering*, 2023.
[147] H. Abukwaik, A. Burger, B. Andam, and T. Berger, "Semi-automated feature traceability with embedded annotations," in *ICSME*, 2018.
[148] W. Mahmood, D. Strüber, A. Anjorin, and T. Berger, "Effects of variability in models: a family of experiments," *Empirical Software Engineering*, vol. 27, no. 3, p. 72, 2022.
[149] S. Greiner, K. Schmid, T. Berger, S. Krieter, and K. Meixner, "Generative ai and software variability - a research vision," in *VaMoS*, 2024.



**Wardah Mahmood** is a doctoral student at Chalmers | University of Gothenburg, Sweden. Her research interests are software product-line engineering, model-based engineering, formal methods, and empirical software engineering. She did her master's degree from Fast National University of Computer and Emerging Sciences, Pakistan in 2016.

**Gül Çalıklı** is a Lecturer (Assistant Professor) in Software Engineering at the School of Computing Science, University of Glasgow in Scotland, United Kingdom. Her research focuses on empirical software engineering, program comprehension, and human cognitive aspects of software engineering. Gül Çalıklı received her Ph.D. degree in computer engineering from Boğaziçi University in Istanbul, Turkey. The papers she co-authored received a Best Industry Paper Award at ESEM'2013, an ACM SIGSOFT Distinguished Artifact Award at ICSE'2020, an ACM SIGSOFT Distinguished Paper Award at ICSE'2021 and an ACM SIGSOFT Distinguished Paper Award at ESEC/FSE 2022. HHe has been a Program Committee member of several leading conferences, including ASE, CSCW, ICPC and MSR. Her service is recognized by a Distinguished Reviewer Award at ICPC'2022.

**Daniel Strüber** is a senior lecturer at Chalmers | University at Gothenburg, Sweden, and an assistant professor at Radboud University in Nijmegen, the Netherlands. His research interests are in model-driven engineering, AI engineering, and variant-rich systems. He was awarded his doctoral degree from Philipps University Marburg, Germany, and worked as a post-doctoral researcher at University of Koblenz and Landau, Germany, and Gothenburg University, Sweden. He is a co-author of over 80 papers with six Best Paper Awards. He has been a Program Committee member of several leading conferences, including ASE, FASE, MODELS. He is the lead developer of Henshin, a model transformation language used in more than 15 countries.

**Ralf Lämmel** is Professor of Computer Science at the University of Koblenz-Landau in Germany since 2007. In the past, he had held positions at Facebook London, the University of l'Aquila, Microsoft, USA, the Free University of Amsterdam, CWI (Dutch Center for Mathematics and Computer Science), and the University of Rostock, Germany. His research and teaching interests include software/data engineering, software reverse engineering, software re-engineering, software language engineering, mining software repositories, program comprehension, functional programming, grammar-based and model-based techniques, and megamodeling. In his latest work at Facebook, he applies machine learning (in a broad sense) in an infrastructural context while developing an increasing interest in data engineering and science. Areas of application concern ownership management, infrastructure simulation, and developer workflow analysis. Ralf Lämmel is one of the founding fathers of the international summer school series on Generative and Transformational Techniques on Software Engineering (GTTSE) and the international conference on Software Language Engineering (SLE). He is the author of Springer textbook on Software Language Engineering: Software Languages: Syntax, Semantics, and Metaprogramming, Springer, 2018, which received the Choice Award "Outstanding Academic Title" in 2019.

**Mukelabai Mukelabai** is a Lecturer in the Department of Computer Science at the University of Zambia. His research focuses on software engineering and quality assurance for variant-rich systems. He received his PhD in Software Engineering at the University of Gothenburg, Sweden in 2022.

**Thorsten Berger** is a Professor in Computer Science at Ruhr University Bochum in Germany. After receiving the PhD degree from the University of Leipzig in Germany in 2013, he was a Postdoctoral Fellow at the University of Waterloo in Canada and the IT University of Copenhagen in Denmark, and then an Associate Professor jointly at Chalmers University of Technology and the University of Gothenburg in Sweden. He received competitive grants from the Swedish Research Council, the Wallenberg Autonomous Systems Program, Vinnova Sweden (EU ITEA), and the European Union. He is a fellow of the Wallenberg Academy—one of the highest recognitions for researchers in Sweden. He received two *best-paper* and two *most-influential-paper* awards. His service was recognized with distinguished reviewer awards at the tier-one conferences ASE 2018 and ICSE 2020, and at SPLC 2022. His research focuses on software product line engineering, AI engineering, model-driven engineering, and software analytics. He is co-author of the textbook on Domain-Specific Languages: Effective Modeling, Automation, and Reuse.