Thesis for The Degree of Licentiate of Engineering

A Framework for Seamless Variant Management and Incremental Migration to a Software Product-Line

Wardah Mahmood

Division of Software Engineering
Department of Computer Science & Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden, 2022

A Framework for Seamless Variant Management and Incremental Migration to a Software Product-Line

Wardah Mahmood

Technical Report No 5084
ISSN0346-718X
Department of Computer Science & Engineering
Division of Software Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2022.

"The value of achievement lies in the achieving."
- Albert Einstein

Abstract

Context: Software systems often need to exist in many variants in order to satisfy varying customer requirements and operate under varying software and hardware environments. These variant-rich systems are most commonly realized using cloning, a convenient approach to create new variants by reusing existing ones. Cloning is readily available, however, the non-systematic reuse leads to difficult maintenance. An alternative strategy is adopting platform-oriented development approaches, such as Software Product-Line Engineering (SPLE). SPLE offers systematic reuse, and provides centralized control, and thus, easier maintenance. However, adopting SPLE is a risky and expensive endeavor, often relying on significant developer intervention. Researchers have attempted to devise strategies to synchronize variants (change propagation) and migrate from clone&own to an SPL, however, they are limited in accuracy and applicability. Additionally, the process models for SPLE in literature, as we will discuss, are obsolete, and only partially reflect how adoption is approached in industry. Despite many agile practices prescribing feature-oriented software development, features are still rarely documented and incorporated during actual development, making SPL-migration risky and error-prone.
Objective: The overarching goal of this PhD is to bridge the gap between clone&own and software product-line engineering in a risk-free, smooth, and accurate manner. Consequently, in the first part of the PhD, we focus on the conceptualization, formalization, and implementation of a framework for migrating from a lean architecture to a platform-based one.
Method: Our objectives are met by means of (i) understanding the literature relevant to variant-management and product-line migration and determining the research gaps (ii) surveying the dominant process models for SPLE and comparing them against the contemporary industrial practices, (iii) devising a framework for incremental SPL adoption, and (iv) investigating the benefit of using features beyond PL migration; facilitating model comprehension.
Results: Four main results emerge from this thesis. First, we present a qualitative analysis of the state-of-the-art frameworks for change propagation and product-line migration. Second, we compare the contemporary industrial practices with the ones prescribed in the process models for SPL adoption, and provide an updated process model that unifies the two to accurately reflect the real practices and guide future practitioners. Third, we devise a framework for incremental migration of variants into a fully integrated platform by exploiting explicitly recorded metadata pertaining to clone and feature-to-asset traceability. Last, we investigate the impact of using different variability mechanisms on the comprehensibility of various model-related tasks.
Future work: As ongoing and future work, we aim to integrate our framework with existing IDEs and conduct a developer study to determine the efficiency and effectiveness of using our framework. We also aim to incorporate *safe-evolution* in our operators.

Keywords

Variant-rich Systems, Software Product-Line Engineering, Variability Mechanisms Process Models, Model Comprehension

Acknowledgment

I am very grateful to my supervisor Thorsten Berger for his resourcefulness and kind support throughout my PhD journey. He has enabled me to compete and publish in top-tier venues in my field, as well as develop links and collaborate with many top researchers. I am also thankful to my examiner Ivica Crnkovic and co-supervisor Gül Calikli for their helpful advice that has always kept me motivated. I am very thankful to my peers at the *EASE* research lab for their continual support and feedback in all major and minor milestones. Specifically, I am thankful to Mukelabai Mukelabai for his tireless guidance and sincere advice. I am also obliged to Daniel Strüber for the patient encouragement and valuable input he always provides. I also extend my thanks to Razan Ghzouli and Ricardo Diniz Caldas for always offering their help and cheering for my achievements. I am very thankful to Leuson Mario, who has relentlessly extended his support regardless of the distance and been encouraging when I needed it the most. I am also grateful to have had the experience to collaborate with many other valuable researchers including Ralf Lämmel, Paulo Borba, Jacob Krüger, Tobias Schwarz, and Christoph Derks. I am fortunate to have the opportunity to work at the Computer Science and Engineering department at the University of Gothenburg, an institute rich with many pioneering researchers, and an environment that fosters professional development. I am extremely thankful for the interactions I have had with my students, indeed I have learned a lot along the way.

I am beyond blessed to have my Father and Mother back home, who have always made sure I feel that the distance is not as much as it seems. I am extremely lucky to have my siblings, Muhammad Hassan Shakir, Fatima Mahmood, Maryam Mahmood, and Khadija Mahmood, as well as Arsal Saeed and Hasher Saeed, all of whom have provided endless support in rough times. Lastly, I am tremendously thankful to my friend Pirah Noor Soomro, who has been my pillar throughout my work, and to Amal Elawad, for providing continual nurturing and support.

This research was partially funded by the Swedish Research Council (2578229 -02), Vinnova Sweden (2016-02804), the German Research Council (SA 465/49-3) and the Wallenberg Academy.

List of Publications

Appended publications

This thesis is based on the following publications:

[A] W. Mahmood, D. Strüber, T. Berger, R. Lämmel, M. Mukelabai "Seamless Variability Management With the Virtual Platform"
Proceedings of the 43rd International Conference on Software Engineering, pp. 1658-1670. 2021.

[B] J. Krüger, W. Mahmood, T. Berger "Promote-pl: a Round-Trip Engineering Process Model for Adopting and Evolving Product Lines"
Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A-Volume A, pp. 1-12. 2020.

[C] W. Mahmood, D. Strüber · A. Anjorin, T. Berger "Effects of Variability in Models: A Family of Experiments"
Accepted at Empirical Software Engineering Journal. 2022

Other publications

The following publications were published during my PhD studies, or are currently in submission/under revision. However, they are not appended to this thesis, due to contents overlapping that of appended publications or contents not related to the thesis.

[a] T. Schwarz, W. Mahmood, T. Berger "A Common Notation and Tool Support for Embedded Feature Annotations"
Proceedings of the 24th ACM International Systems and Software Product Line Conference, pp. 5-8. 2020.

[b] W. Mahmood, M. Chagama, T. Berger, R. Hebig "Causes of Merge Conflicts: A Case Study of ElasticSearch."
Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems, pp. 1-9. 2020.

[c] L. Da Silva, P. Borba, W. Mahmood, T. Berger, J. Moisakis "Detecting Semantic Conflicts Via Automated Behavior Change Detection."
IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 174-184. 2020.

Research Contribution

In this section, I summarize my contributions to the appended papers. The contributions are classified using the Contributor Roles Taxonomy (CRediT)*.

In Paper A, I led the analysis of the state-of-the-art frameworks, as well as the conceptualization, formalization, and implementation of our framework. Additionally, I integrated the Git commands' execution from the IDE and implemented three parsers (JavaScript, Java, JSON) for evaluation. I led the simulation study for the evaluation as well. I also participated equally in writing the original draft of the paper, and led the review and editing of the paper. I also designed the online appendix of the paper.

In Paper B, I participated in the analysis and classification of publications, as well as the design of the simplified process model. I participated in writing two sections of the paper, and also in the review and final editing of the camera-ready version of the paper.

In Paper C, I led the design and analysis of the third experiment. I also led the write-up of the last two experiments. I designed the materials for the experiment and performed data pre-processing to facilitate the experiment. I reused the analysis scripts from the first experiment. I also designed the online appendix of the paper. Lastly, I led the proofreading and editing of the camera-ready version of the paper.

*https://casrai.org/credit/

Contents

Chapter 1

Introduction

With the advent in digitization, software systems prevail in virtually all domains and disciplines. Customers of software systems have varying needs and expectations, and as a result, software systems need to exist in many variants. These variants tend to a variety of expectations regarding quality, functionality, performance, and usability etc. Additionally, they address a diverse set of market segments and need to function on different run-time environments and hardware systems. Companies cater to these varying expectations by creating a portfolio of systems which share considerable functionality with each other. These variant-rich systems differ in terms of the *distinguishing* features they implement.

Developers mainly realize variant-rich systems using two contrasting approaches. One approach is ***clone&own***, in which developers create a variant and then later clone it and adapt it to create another variant. Developers adapt variants in response to various stimuli, such as new customer requirements, innovative feature introduction, and variant tuning for different hardware platforms etc. This strategy is strongly favored by developers due to its inexpensiveness, agility, and ease of adoption. It also offers independence to developers and does not impose any additional training efforts. Additionally, clone&own can be conveniently deployed at any granularity level, e.g., cloning an entire project vs. cloning a block of code. Similarly, cloning can be performed at any stage; requirements, design, implementation and testing. Clone&own is well-supported by popular version control systems such as GIT* and Mercurial† owing to their forking, branching, merging, and pull request facilities.

Despite its many benefits, clone&own does not scale well when the number of variants increases. Clones of *source* (or cloned) variants lose their link with the source variant as soon as they are cloned. Subsequently, the variant and its clones evolve independently. However, some changes, such as bug-fix propagation, feature introduction and feature enhancement etc. often need to be applied across variants. To replicate the changes across variants, developers manually traverse the code-base of the variants to find the clones (a.k.a *clone detection*) of the changed artifacts (henceforth referred to as *assets*) and apply the changes. Consider the example of a mobile phone company which uses

*`https://github.com/`

†`https://www.mercurial-scm.org/`

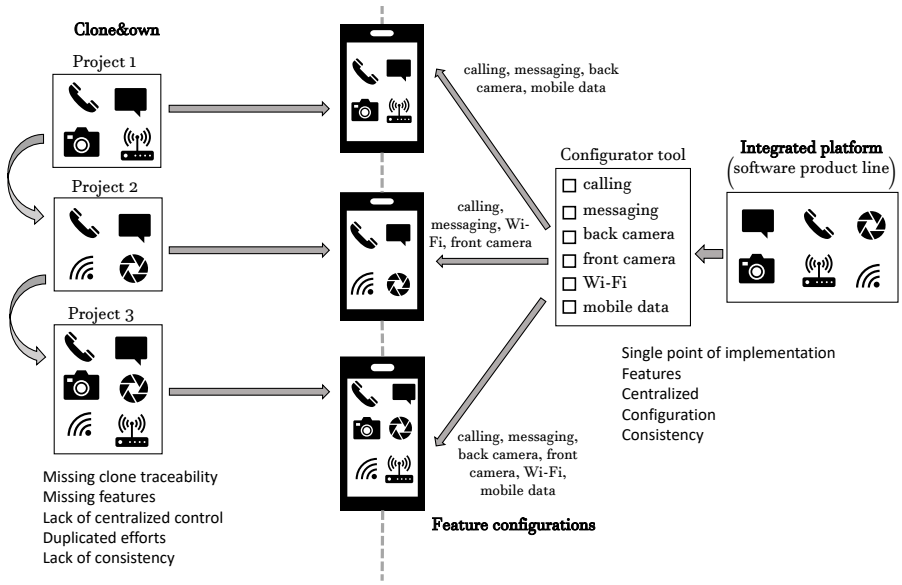


Figure 1.1: Two contrasting approaches for realizing variant-rich systems: clone&own (left) and software product-line engineering (right)

clone&own to create mobile phone variants (Figure 1.1, left side). In project 1, developers decide to update the "calling" functionality to also support *video calling*. During the testing of the feature with the end users, developers realize that the feature is well-received, and that it adds value to the system. Developers now wish to reflect the same update in other projects (*change propagation*). To achieve this, they have to traverse the code base of project 2 and project 3, and find clones of assets which implement "calling". Moreover, if the updated feature is scattered across multiple software assets, developers have to manually find the different locations in project 2 and project 3 where the feature was implemented (a.k.a *feature location*). Even determining the right source variant for cloning, or determining the target variant for change propagation, is challenging, since it is often far from obvious what features are realized in a variant. Both clone detection and feature location are typical and frequent developer tasks which can be performed either manually or using third-party tools. Manual clone detection and feature location is laborious and time-consuming, and can affect developer productivity. Developers can also use third-party tools, however, such tools mainly rely on heuristics for detecting clones and locating features, and are therefore inaccurate at best. Moreover, using these tools requires extra setup effort as well as fulfilling the assumptions and preconditions of the tool (e.g., compliance to specific input formats or providing a seed value).

An alternative approach for realizing variant-rich systems is creating a configurable, fully ***integrated platform*** by deploying platform-oriented engineering methods, such as software product-line engineering. An integrated platform (henceforth referred to as a software product-line) is a holistic containment of all features implemented and tested only once (see integrated

platform, Figure 1.1). Individual variants are derived from the platform by providing *valid* feature configurations. This approach suits best to systems with many variants, such as software product lines (e.g., automotive/avionics control systems and robotics) and highly configurable systems (e.g., Linux Kernel and Marlin 3D printer). In contrast to clone&own, this approach scales well when the number of variants increases. A software product-line offers centralized control; changes such as bug-fix and feature enhancement only need to be implemented once in the platform. Changes are automatically reflected in the subsequently configured variants. A software product-line offers higher consistency in products, as they are derived from a unified platform. Additionally, software assets in a product-line are planned for reuse, leading to a higher focus on quality assurance of these shared and reusable assets.

Software product-line engineering offers better reuse and maintenance, however, adopting a product-line approach poses a few overheads. It can be expensive and time-consuming, requiring extra steps such as domain analysis and scoping[‡], variability analysis[§], and core asset implementation and testing. These additional steps lead to a delayed initial time-to-market. Starting with a fully integrated platform from scratch also requires incorporating variability-specific concepts, such as feature models, variability mechanisms, and configurator tools to make assets reusable and configurable. Research shows that while product-line adoption can be expensive, the high upfront investments soon pay off. In fact, companies reach a break-even point with as few as three variants (Figure 1.2), with dramatic cost reductions per variant in comparison to single-system development [2].

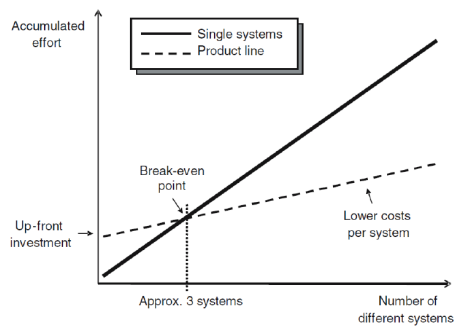


Figure 1.2: Economics of software product-line adoption from [2]

In practice, organizations rarely build a fully integrated platform from scratch, but rather start with clone&own, owing to its ease of adoption. Another reason for inclining towards clone&own is that organizations typically do not have specialized personnel for planning reuse [3], leading them to incline towards the readily available solution; cloning. However, with clone&own, they face inevitable maintenance overheads in the long run. With an ever-increasing number of variants, developers lose overview over the products. Soon, simple

[‡]Involves studying the domain to determine the intended specifications and scope of the project

[§]Analyzing the alternate implementations of various features, e.g., operating system of a mobile phone (iOS Vs. Android Vs. Windows)

changes such as feature updates and bug fixes become very difficult to propagate. Consequently, a migration from a lean architecture to a software product-line one is executed, which in itself is challenging. An architecture migration involves major code refactoring, making it risky and effort-intensive. In addition, it poses cost overheads, and relies on recovering important metadata that was never recorded during development, such as the location of features in assets (feature location), and traceability between clones (clone detection). A classic example of architecture migration to a product-line one is the case of Danfoss [4], an electronics manufacturing company, where migration took four years, and required experts both from inside and outside the organization, in addition to many organizational changes.

To overcome the aforementioned challenges, researchers have contributed a few frameworks for product-line migration and variant synchronization (change propagation). These frameworks follow an *extractive* strategy to recover metadata from a set of given variants. The migration frameworks, as we will show (in **RQ1**), are either too abstract, or predominantly rely on heuristics techniques for feature identification, feature location, clone detection, and variability mining. Unfortunately, the heuristic techniques are usually not accurate enough to be applicable in practice, and require substantial effort to set them up and provide with manual input (e.g., specific program entry points for feature location techniques). Furthermore, the migration frameworks prescribe a non-iterative, "single-hit" strategy, which is risky and expensive. The variant synchronization frameworks partially solve the change propagation problem. However, the frameworks either do not have the notion of features altogether, still requiring developers to perform feature location frequently, or only support change propagation in code-level assets.

The overarching goal of this PhD is to bridge the gap between clone&own and a fully integrated platform in an accurate, risk-free manner. We aim to provide a non-restrictive method of working, in which developers conveniently switch between the two extremes and reap the benefits of both. To work towards this goal, we begin by understanding the relevant literature to survey the capabilities and shortcomings of the existing frameworks. We qualitatively evaluate various frameworks supporting either variant synchronization or architecture migration to a software product-line. Specifically, we study five state-of-the-art frameworks and compare them against our framework (presented in **RQ3**). To this end, we investigate the following research question:

RQ1: *What are the state of the art frameworks for software product-line migration and variant synchronization?*

To answer this research question, we iterate through the literature to gather relevant frameworks. We identify five related frameworks, three of which focus on product-line migration [5–7] and two on variant synchronization [8, 9]. We report our findings from RQ1 in Paper A.

Our findings suggest that although many researchers have attempted to address the issues prevalent in software product-line migration and variant synchronization, a number of factors hinder the applicability of their contributions in practice. The frameworks for product-line migration all prescribe an extractive strategy; recovering metadata pertaining to clones and features heuristically. The framework by Rubin *et al.* [5, 10] comprises a rather abstract set of operators which rely on heuristics (e.g., for feature identification and

feature location) and third-party tools (textual diff tools for clone detection). The framework by Fisher *et al.* [6] is concrete and operational, but yet again, relies on heuristics for feature identification and location, and clone detection. Martinez *et al.* [7] also propose a framework for variant integration, which although usable, heavily relies on the involvement of product-line adopters and domain experts, rendering it laborious and error-prone.

The frameworks for variant synchronization are also limited in their capabilities. VariantSync, a plugin developed by Pfofe *et al.* [8] supports automated variant synchronization of related variants. VariantSync significantly reduces inaccuracies posed by manual or tool-assisted clone detection and feature location, however, it is only applicable at the lowest level: actual code. In reality, assets of different types, including repositories, directories, files, and classes can require synchronization. Montalvillo and Díaz [9] define branching models to support change propagation between core assets (assets belonging to all variants) and variant-specific assets (assets belonging to some variants). The framework is effective, however, it is limited because of frequent merge conflicts. Additionally, without the notion of features, when the number of variants increases, it becomes difficult to establish source and target variants for variant synchronization. We summarize the discussed frameworks and provide a qualitative comparison against our framework in Section 1.4.

After establishing an overview of the relevant literature, we proceed to explore the dominant practices adopted by software product-line engineering practitioners in industry. Our goal is to understand the state-of-the-art, and determine if the practices actually reflect the process models for product-line adoption prescribed in literature. Following, we briefly look into the prevalent industrial practices regarding software product-line adoption and evolution. Accordingly, we investigate the following research question:

RQ2: *What are the contemporary product-line engineering processes employed in practice? Do they accurately depict the process models for product-line adoption presented in literature?*

To investigate this research question, we begin by surveying the most prominent process models for product-line adoption. These process models [11–15] steer the adoption and evolution of product-line engineering in industry and as such, are considered the de-facto standard for incorporating platform-oriented approaches in any organization. Next, we survey the contemporary practices employed in industry by analyzing the experience reports reporting product-line adoption and evolution in industry. To this end, we gather publications reporting experiences with product-line adoption published in the past five years. The aim is to determine if there are discrepancies between the process models from literature, and the practices adopted in industry.

Notably, most process models mainly focus on the scenario of *proactive adoption*; implementing a product-line architecture from scratch. Proactive adoption however, as explained above, relies on conducting intensive domain analysis and scoping before the realization of a centralized platform, delaying the initial time to market. In reality, developers prefer to capitalize on their existing variants to create new ones. Secondly, the process models prescribe a clear distinction between the two phases of software product-line engineering:

domain engineering (DE)[¶] and application engineering (AE)[‖]. In reality, the distinction is not very strict. Organizations can *extract* a platform from their existing variants. Moreover, developers can still implement changes in the platform (DE) even after some variants have been created (AE). So, while most process models assume a sequence between domain engineering and application engineering, these phases go hand in hand in reality. Lastly, most process models assume that product lines only evolve through their platforms; variability is known and realized at the time of platform construction. In practice however, while some variability is known prior to platform construction, it is common for platforms to evolve via variants as a result of new customer requirements and hardware adaptations etc.

We gained a number of insights from our analysis of RQ1 and RQ2. First, industrial cases of software product-line adoption and evolution do not mirror process models prescribed in literature. Most assumptions in process models are vague and obsolete, and consequently, need to be updated to reflect actual scenarios and challenges developers face when undertaking Software Product-Line Engineering (SPLE). We present an updated process model, ***promote-pl***, which we synthesize from industrial reports of product-line adoption in literature. Secondly, while existing frameworks and process models have attempted to steer product-line adoption, adopting SPLE remains a demanding and risky endeavor. Most approaches require halting the development to proactively, reactively[**] or extractively[††] create an integrated platform, requiring developers to perform tasks they are not familiar with: scoping, variability mechanism incorporation, feature modeling, and configuration etc. Thirdly, when creating variant-rich systems, organizations choose either clone&own or software product-line engineering and adhere to it. A switch is only executed if the organization is struggling with reuse or maintenance. The switch is mostly "single-step", involving code refactoring, developer intervention, and third-party tools. The adoption approaches discussed above assume that once a platform is in place, organizations will proceed with the platform, and cease using clone&own. In practice, developers still continue to use clone&own to their convenience even after adopting a product-line architecture.

We take a different route than the existing frameworks and process models, and provide a non-restrictive, non-binary mode of working. We conceive an approach which exploits the spectrum between clone&own and software product-line, offering both variant management and product-line migration. To this end, we prescribe *truly incremental migration*; starting from cloned variants, and building incrementally towards a fully integrated software product-line. Following, we investigate how to formalize truly incremental migration.

RQ3: *How to design an appropriate framework to realize the truly incremental migration. Specifically, what operators and conceptual structures need to be defined, and how?*

As mentioned above, our framework supports truly incremental migration. Specifically, we take-over the 6 governance levels proposed by Antkiewicz *et*

[¶]Involves scoping the product-line and building the core-assets

[‖]Involves implementing variant-specific functionality

[**]Reactive adoption starts with one variant, which becomes the platform. All subsequent variants are merged into the platform one by one.

[††]Extractive adoption starts with n variants, which are migrated to a platform in a single-step after extensive commonality and variability analysis.

al. [1]—each level representing the system at an increased stage of maturity. Instead of recovering metada about clones and features, we proactively store metadata incrementally using simplistic data structures. Each governance level adds further detail to the system and in return, offers incremental benefits. Our framework is multi-step and minimally invasive, rendering it less prone to risks. Additionally, by proactively storing metadata, we eliminate inaccuracies resulting from reliance on developers and third-party tools. The stored metadata can be conveniently used for change propagation. Additionally, the metadata enables *feature cloning*, a novel use-case that supports cloning entire features across variants. In addition, it can be exploited to integrate variants, and even migrate variants to a product-line, if needed. In the event that an organization does not desire to adopt a product-line architecture immediately, the metadata can still be maintained as a "safety net" to facilitate any planned or needed architecture migration to a product-line one in the future.

We devise a framework called "virtual platform" for incremental migration, which generalizes various variant management and product-line migration frameworks. Akin to the framework by Rubin *et al* [5,10], our framework is also *operator-based*; offering operators that bridge the gap between clone&own and software product-line architecture. While their framework comprises abstract operators, our operators are concretely implemented and tested. Moreover, in contrast to the frameworks by Fischer *et al.* [6] and Martinez *et al.* [7], our operators do not rely on being invoked in a specific order.

The foundation of our framework lies in a set of *conceptual structures*, which are semi-structured representations of code in memory. The conceptual structures capture the semantics of the code, but are orthogonal to the code syntax, making our framework language-independent. The *sanity* of our conceptual structures is governed by a set of *well-formedness constraints*, which are predefined rules that look over the operations over the structures. These conceptual structures are used for two main purposes. First, they are used to maintain an in-sync copy of the variants and the various assets they comprise (directories, files, classes etc.), and as such, form a foundation for invoking our operators responsible for variant management and product-line migration. Second, they are used to store two important kinds of metadata: clone traceability and feature-to-asset traceability (referred to as feature mappings). After establishing a high-level overview of the entities our framework will comprise, we capture the relevant details for all conceptual structures, and develop a *conceptual model*. To this end, we iteratively discuss the entities and the relationships among them, and the attributes of each entity that are relevant to our framework. We aim for a good balance between usefulness and required memory space. Our conceptual model comprises of assets and features, among other entities.

On top of the conceptual structures, we devise *operators* that manage and manipulate these conceptual structures. Virtual platform offers a multitude of operators for managing and exploiting the conceptual structures. The operators which manage the conceptual structures are responsible for synchronizing these structures with the working copy of the variant, and for augmenting the conceptual structures with metadata. As explained above, we store two kinds of metadata: clone traceability and feature-to-asset traceability. We store clone traceability *silently*; without requiring explicit invocation by the developer.

In contrast, feature traceability is stored proactively by the developers. We also provide operators for exploiting the stored metadata. Specifically, we offer support for querying the conceptual structures to retrieve information such as assets containing the implementation of a feature (feature location), and all clones of a particular asset (clone detection) etc. This information becomes relevant in other super-tasks, which have their own dedicated operators, such as change propagation from original assets to their clones and vice versa. When designing operators, we focus on inputs, tasks and required outcome of each. A summary of our conceptual structures and operators is discussed in Section 1.4. Next, we investigate the following research question:

RQ4: *What is the effectiveness of the framework virtual platform? Specifically, what are the costs and benefits?*

We implement a prototype of our framework in Scala. Parallel to implementation, we test each operator using realistic scenarios inspired by literature and experience. We evaluate our framework both qualitatively and quantitatively. For the former, we conduct a comparative assessment of our framework against the frameworks analyzed in *RQ1.* For the latter, we perform a cost and benefit analysis of using virtual platform on a real-world open-source system having four variants. Since our approach relies on explicitly recorded feature-related metadata, we choose a case study with explicitly recorded features [16]; Clafer WebTools[‡‡]. The variants are originally realized using clone&own, and after forking, there is significant cloning in the variants throughout development. We simulate the development of the variants by retrofitting our operators to the operations performed in every commit, e.g., addition of a file or modification of a method.

As primary validation, we verify that our operators suffice to cover each evolution activity performed in the actual variants (e.g., addition of a file). We observe that the virtual platform has a corresponding operator to each evolution activity. Secondly, we compare two costs: the *added* cost and the *saved* cost. Added cost is the additional workload on the developers to invoke the operators. This mainly constitutes the operators for recording feature-related metadata. Saved cost is the time taken for clone detection and feature location that the developers are able to save. These costs are saved because the explicitly recorded metadata eliminates the need to locate clones and features; developers can query the virtual platform to retrieve this information. Based on our analysis, developers reach a break-even point if they take 54 seconds to record *one* feature. In practice, features exist in the developers' subconscious when they are developing those features, and as such, documenting them takes significantly less than 54 seconds. We envision better accuracy when virtual platform is used alongside development. A summary of our evaluation and results is presented in Section 1.4.

We already established that explicitly recorded features can be exploited for a variety of purposes including automated feature location and feature cloning. Next, we investigate their potential beyond code-related tasks; we explore how features can be used to improve model comprehension. To this end, we look into the following research question:

RQ5: *Can features be leveraged to facilitate model comprehension?*

[‡‡] `https://github.com/gsdlab`

Thus far, we explore how feature mappings can be exploited to facilitate *code* reuse and maintenance. Researchers have also attempted to study the impact of different variability mechanisms on the *comprehensibility* of code [17]. Code comprehension is often a prerequisite to many tasks pertaining to software evolution and maintenance. While code is mostly syntactic and technical, features in code add a high-level perspective to the software and facilitate code comprehension.

As organizations streamline their processes for developing variant-rich systems, they realize that variability needs to be incorporated in all key development artifacts, including models. As a result, researchers have invested their efforts in developing variability mechanisms for models. Variability mechanisms are available for both UML [18–20] and Domain-Specific Modeling Languages (DSMLs) [19, 21, 22]. Models are often used to steer development, and offer an abstract means of analyzing the system before it is implemented. They serve many purposes including code generation [23] and automated verification [24]. While the impact of variability mechanisms on code comprehension has been studied [17], so far, the impact of these mechanisms on model comprehensibility is not empirically investigated. We investigate the impact of different variability mechanisms on the comprehensibility of *models*. We conduct an empirical study comparing the efficiency of using two popular variability mechanisms: *annotative* and *compositional* variability on the efficiency of routine model comprehension tasks developers perform. Annotative variability involves adding features *inside* models by annotating various model elements with feature annotations. Compositional models comprise various model fragments, each fragment representing the structure or behavior of the software for one feature (or feature combination). The integrated model is a composition of model fragments pertaining to the features implemented in the variant. In our analysis, we consider three model types: class diagrams, state machine diagrams, and activity diagrams.

We report our findings from a family of three experiments, each featuring a different model type. For each experiment, we design tasks of three types: understanding variants, comparing two variants, and comparing all variants. As participants, we recruit 164 (BSc, MSc, PhD) students from various universities. Participants are required to perform tasks pertaining to the task types mentioned above. We follow the *Latin-squared design* [25, 26], such that each participant experiments with each variability mechanism and subject system once. After the participants have interacted with each variability mechanism, we ask them for their subjective assessments and preferences. We evaluate the *efficiency* of the variability mechanisms using two metrics: *correctness* and *time* taken to complete the tasks. Our findings indicate that in majority of the cases, annotative mechanism leads to better performance in terms of correctness and time taken. Additionally, annotative mechanism is favored over the compositional one in majority of the task types and experiments owing to its conciseness. Based on our analysis of the qualitative responses, we also discover that the choice of the variability mechanism depends on a number of factors including scalability, overview, flow, efficiency, and intuitiveness etc. We present the details of our methodology in Section 1.2, and the results of our analysis in Section 1.4.

This report is structured as follows: we present a background of relevant

terminology and concepts in Section 1.1. We discuss the methodology we adopt to answer each research question is Section 1.2. We summarize our publications in Section 1.3. We answer our research questions in Section 1.4. We conclude the contributions of our work and discuss our ongoing and future work in Section 1.5. We append our publications as Chapter 2, Chapter 3, and Chapter 4.

1.1 Background and Motivation

In this section, we define the terminology used in this report. We also elaborate on clone management, variant management and software product-line migration in variant-rich systems. We explain the relevance of features in model comprehension. Lastly, we discuss the governance levels presented by Antkiewicz *et al.* [1], which drive the design of our framework.

1.1.1 Terminology

- **`Asset:`** Asset is any artifact belonging to a variant (explained shortly). Assets can be classified into different kinds depending on their content: requirement documents, models, code, and test cases etc. Assets can also be of different *types*, depending on their level of granularity in the project: repository, directory, file, class, method, and block (of code or text). Code assets contain partial or complete implementation of features (explained shortly) or feature combinations. In traditional clone&own, assets of different types are cloned from one variant to another.

- **`Variant:`** A variant is a customized, stand-alone instance of the system, either realized via clone&own, or derived from a centralized platform (explained shortly). Variants comprise assets of different types, each implementing one or more features. Variants can also comprise requirements and design documents, system analysis models, test cases, and user interface screens etc. Variants correspond to feature selections, and one variant differs from another based on the distinguishing features it implements.

- **`Feature:`** Features are distinctive high-level abstractions of functionality relevant to one or more stakeholders. Features can be fine- or coarse-grained, but they are generally more abstract and less technical than the code itself. Features offer a non-technical perspective to the functionality and facilitate communication between developers. Features are derived from requirements, and may or may not be documented explicitly. Research shows that explicitly documenting features leads to better reuse and maintenance [16] and improves the accuracy of program comprehension [17].

- **`Feature Model:`** Feature models are hierarchical, tree-like structures composed of features. These models capture the relationships among features. Additionally, they also capture dependencies and constraints between different features in the same feature model. Feature models allow decomposing functionality into small, meaningful units (features). Once incorporated, they can be used to check validity of configurations (explained shortly) when deriving variants.

- **`Variability:`** Variability is the ability of a platform to comprise alternate implementations of the features it comprises. Variability allows variants and assets to be customized, extended, and configured to fit certain requirements.

- **`Variability mechanism:`** Variability mechanisms are ways in which variability can be incorporated in software assets. This mainly involves *mapping* assets to the features they implement (e.g., `#ifdefs` in the Linux kernel [27]). Once a variability mechanism is in place, the assets can be made configurable, and developers can derive variants by providing valid feature configurations (explained shortly).

- **`Configuration:`** Configuration is the process of composing a variant given a feature selection. Feature selection is retrieved from customers, and checked against the feature model for validity; a *valid* configuration conforms to the relationships and the constraints among the features, and holds the dependencies among various features. For valid configurations, assets implementing the selected features are picked and added to a newly created variant.

- **`Platform:`** A platform is a holistic containment of all features implemented once. Variability is known and implemented in the platform by allowing variable implementations of a feature to provide customization. A platform offers centralized control, and offers better reuse and maintenance. In addition, it allows configuration; using a configurator tool, developers can *derive* variants corresponding to various customer requirements.

1.1.2 Clone Management in Variant-rich Systems

As explained above, organizations often deploy clone&own as a reuse mechanism to quickly create and deliver new variants to the market. While cloning is a convenient and readily available solution, the reuse comes at the cost of effort duplication in the long run. Due to lack of any centralized control, developers soon lose overview over the variants. This impedes subsequent reuse of the variants, as with the increasing number of variants, it becomes increasingly difficult to establish which variant to clone to create a new one. Maintenance of variants also becomes exceedingly difficult when the number of variants grows. For routine changes such as bug-fixes and feature enhancements, developers have to manually locate all places where the assets were cloned to replicate those changes manually. Without an explicit clone management framework [5–7, 9] or tool [8, 28] in place, these tasks become daunting in the long run, and hinder developer productivity. Even with the presence of a framework or tool, developer involvement is inevitable; the frameworks and tools require significant setup effort and training.

Virtual platform provides accurate and efficient clone management by recording and exploiting two kinds of metadata explained above. With *clone traceability*, the daunting and error-prone task of clone detection becomes accurate and automated. To propagate changes, virtual platform can be queried to *get* the clones of the changed asset, and automatically propagate these changes to the clones. With *feature mappings*, establishing which variant to clone (reuse) to create another one is easier, as features provide a higher level of abstraction, and developers do not need to understand the code inside the variants to find out which features they implement.

1.1.3 Variant Management in Variant-rich Systems

In a fast and competitive market, the need to deliver to rapidly evolving customer requirements intensifies. As a result, organizations deploy *variant management* strategies to deliver customer-tailored solutions quickly to the market. Variant management involves implementing processes and product models to streamline the process of variant creation and deployment. Variant management can be approached in a number of ways. The simplest strategy is by adopting clone&own. Clone&own, as explained above, allows quickly creating new variants in response to incoming customer requirements, and introducing them to the market. With no means of governing the number of variants, this customer-focused development strategy can in the worst scenario, lead to one special solution per customer. This results in a range of maintenance and reuse problems, as explained above.

Other strategies for variant management include independent component teams [29], platform versions [30], and model-based variant management [31] etc. Ideally, customized variants should be a composition of stable, standalone components, which are modular solutions of (one or more) features the variant implements. The components once standardized, offer a high degree of flexibility. Software product-line engineering is a systematic approach for variant management, which allows software systems to be decomposed into features. The features are independently built in standalone components (a.k.a modules), and tested individually and in combination with other features. This approach heavily contrasts ad-hoc clone&own, which focuses on (non-systematic) *variant reuse* instead of (systematic) *feature reuse*. Clone&own allows convenient reuse but leads to difficult maintenance. SPLE lies on the other extreme of the *spectrum*, where the initial development can be daunting and labor-intensive, but the reuse and maintenance is relatively effortless.

However, as explained in Section 1, implementing SPLE is a demanding and time-consuming endeavor. SPLE involves planning for reuse, which requires specialized personnel including domain experts and third-party tool practitioners (e.g., for incorporating variability). SPLE also requires developers to perform tasks unfamiliar to them, such as using variability mechanisms and making assets configurable. These factors significantly slow the variants' initial time to market, and pose the risk of falling behind in a competitive market.

Virtual platform offers a middle ground between ad-hoc clone&own and SPLE by exploiting the spectrum between the two. Firstly, it allows developers to *incrementally* transition from a set of variants realized via clone&own to a fully integrated software product-line. By design, our framework is risk-free, avoiding a single-step migration, and relying on accurate metadata. Recording feature-to-asset mappings (e.g., using embedded annotations, Paper a [32]) enables configuring assets and facilitates incorporating an SPL architecture. Secondly, our framework is not restrictive; allowing developers to still use clone&own even after having a platform in place, and continuing to record metadata for integrating the clones into the platform. In the event where an SPL architecture is never realized, virtual platform still provides automated change propagation and variant management by exploiting the metadata pertaining to clones and features.

1.1.4 Software Product-line Migration in Variant-rich Systems

Aiming for efficiency and agility, organizations typically start creating a portfolio of variants by reusing their existing variants; using clone&own instead of investing in a product-line architecture. As the portfolio grows in size, developers run into a range of reuse and maintenance challenges (as explained above). To overcome those challenges, organizations plan to invest in fully integrated platforms. Having a platform allows developers to focus on developing high-quality assets, which can then be configured to be included in different variants. This "migration" to a product-line architecture can be approached in different ways.

Reactive adoption allows developers to migrate to a platform incrementally: one variant at a time. The first variant is the platform itself, and all next variants are integrated sequentially into the platform as it grows. While reactive adoption is a safe approach owing to its incremental nature, it is limited by accuracy. Every time a variant is merged into the platform, the *commonality* (shared functionality with other variants) and *variability* (distinguishing functionality of the merged variant) need to be established. This involves developer intervention, either manual or tool-assisted. Moreover, since the feature traceability was never recorded, developers have to locate features inside assets, and record the traceability. Once the features are known and automated, significant effort is required to incorporate variability mechanisms and make assets configurable. An example of reactive adoption is the ECCO framework [6]. Another approach for migration is *extractive* adoption, which merges multiple variants at once into a platform. Extractive adoption involves performing a commonality analysis to identify clones of various assets, and performing feature identification and location for all features implemented in different variants. Once the clone traceability and feature-to-asset traceability are recovered, the variants are integrated, and the assets are made configurable. This approach also heavily relies on developers and third-party tools for clone detection and feature location, and is prone to risks and inaccuracies. Additionally, the "single-step" nature of extractive adoption can in worst case require organizations to halt variant development, and focus solely on the architecture migration.

Virtual platform offers smooth, risk-free migration by allowing developers to conveniently record metadata, and then relying on the metadata for accurate clone detection and feature location. In this way, developers continue to perform tasks they are accustomed to, while only recording information of the features they implement. Clone traceability, as explained above, is recorded silently in the background automatically. Moreover, the migration is incremental, such that each level adds more metadata and provides incremental benefits. Organizations can choose to implement only a subset (or all) of the governance levels depending on their needs, and realize the benefits the levels provide.

1.1.5 Model Comprehension in Variant-rich Systems

Models are high-level abstractions of the software, and play a significant role in development and testing of software systems. Models offer a simpler view of the system, and allow analyzing the system before it is realized. In the context

of variant-rich systems, the need for models intensifies, as such systems are inherently complex and tend to a variety of customer requirements. To fully explore the capabilities of models in various stages of the development life cycle, it is important that they are understood well. Model comprehension is the task of understanding models that represent the structure or behavior of the software systems. Researchers have attempted to empirically investigate the impact of using different types of models on the accuracy of model comprehension. The model types include graphical and tabular models [33, 34], UML models [35], and textual languages [34].

With an increasing focus on reuse-oriented development processes, software product-line engineering is gaining popularity in industry. As a result, variability mechanisms are devised to allow practitioners to incorporate variability in code, and more recently, models. Variability mechanisms, as explained above, are used to add feature traceability in software assets. Features are non-technical abstractions over code, and often act as a perceivable mode of communication between clients and developers. A study on the impact of different variability mechanisms on *code* comprehensibility shows that feature annotations have a positive impact on code comprehension [17]. Another empirical study by Santos et al. [36] shows that there is no difference between the correctness and response time of code comprehensibility using feature-oriented programming (FOP) and conditional compilation (CC). So far, the impact of using different variability mechanisms on the comprehensibility of models is not systematically investigated. We conduct the first empirical study to investigate the impact of using two pupular variability mechanisms: annotative variability and compositional variability, on the efficiency of model comprehension.

We report our findings from a series of three experiments, each featuring a different model type. We use different subject systems in each experiment to aim for better generalizability. We detail the methodology of our experiments in Section 1.2 and present our results in Section 1.4.

1.1.6 Governance Levels by Antkiewicz *et al.* [1]

Antkiewicz *et al.* [1] present six governance levels spanning from clone&own to a fully integrated platform. The levels set out truly incremental migration; transitioning from a lean, clone-based architecture to a reuse-oriented platform based one in a progressive manner. Figure 1.3 represents the governance levels, with the benefits of incremental migration on the left, and the drawbacks of non-systematic reuse on the right. Next, we briefly discuss each governance level, focusing on the problem it solves, the solution it provides, and the metadata it saves (if any). It is important to note that the first level (L0) is not fundamentally one of the governance levels defined by Antkiewicz *et al.* [1]. However, we still discuss L0 to build motivation for the subsequent levels.

Problem: Organizations need to quickly respond to varying customer requirements and deliver variants.

L0. Clone and own: They deploy clone&own, which is convenient and readily available. Developers traverse through the existing variants to find a suitable one to clone; the functionality of which is closest to the newly requested variant.

Approach: Developers clone a variant, adapt it for new requirements, and

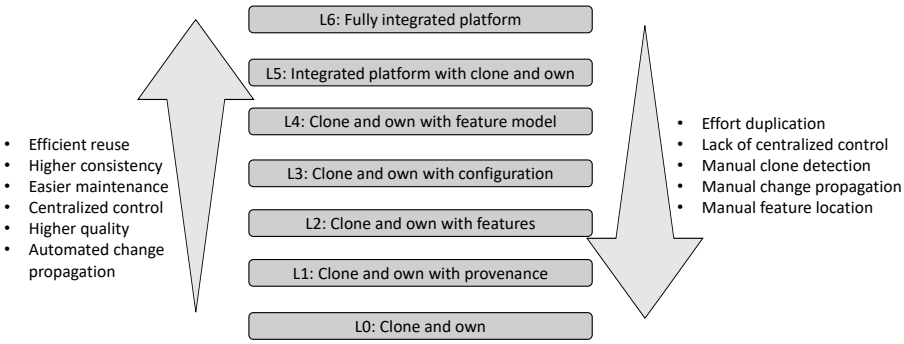


Figure 1.3: Governance levels by Antkiewicz *et al.* [1]; L0-L6 represent the system at increasing stages of maturity

deploy it to the customer.

Problem: With rapidly changing customer requirements, organizations continue creating new variants non-systematically. Consequentially, the number of variants grows exponentially. To maintain the code, simple tasks such as propagating a bug-fix, become increasingly difficult, mainly as the organization lacks a way to govern where the clones of different variants are.
L1. Clone and own with provenance: Organizations resume cloning, but also record metadata about clones to enable tracking clone traceability. The benefits are mani-fold. The metadata can be used to synchronize variants (change propagation) and improve consistency among variants. Additionally, it can be used to integrate multiple variants into one if the organization aims to transition to a product-line architecture.
Approach: Developers incorporate a way to store metadata about clone traceability. This can be achieved in many ways, e.g., textual files and external databases etc. On top of the metadata, developers provide ways to traverse through the stored information and query the system (i.e., virtual platform) to get information about clones.

Problem: Developers lose overview over variants. It gets increasingly difficult to determine which features are implemented in a variant. Consequently, it is laborious to determine which variant to clone when creating a new one, as developers have to traverse the variants to find the features they implement. Additionally, tasks such as enhancing the functionality of a feature, or cloning a feature across variants become increasingly demanding.
L2. Clone and own with features: Developers add the information about features the variants implement. Specifically, they record which assets implement a feature. This eliminates the effort needed for feature location, and adds a non-technical overview over the variants. Additionally, having explicitly recorded features enables feature-oriented reuse. Feature-oriented evolution also becomes efficient, as the recorded feature-to-asset traceability makes finding and modifying features convenient.
Approach: Developers record feature-to-asset mappings either internally or

externally to the software assets. Since features already exist in the subconscious of the developers, the only effort required is for materializing the information in physical memory.

Problem: Despite explicitly recorded features, due to cloning, there still is substantial redundancy in the variants.
L3. Clone and own with configuration: Developers make the assets configurable; on top of the feature-to-asset mappings, developers incorporate variability mechanisms, so that developers can enable or disable features and control variation points.
Approach: Developers maintain a list of features, and incorporate a variability mechanism of their choice to add feature mappings (e.g., embedded annotations, Paper a [32]). Additionally, they add a configurator, which generates a variant given a feature selection.

Problem: With time, it becomes difficult to maintain an overview over the features, especially if the variants grow in size. Additionally, features can have dependencies and constraints that need to be maintained in order to generate valid variants.
L4. Clone and own with feature model: Developers structure the maintained list of features into a feature model. The feature model is dynamic; it can grow and shrink at any time during the development of the variants. Developers can use the feature model as a non-technical artifact for communication, and govern the valid configurations by providing the feature model as an input to the configurator tools.
Approach: Developers create a feature model by defining the relationships, dependencies and constraints among features. To this end, they either analyze the code from the variants to determine how various features interact with each other (bottom-up), or extend the domain analysis to identify relationships among features (top-down).

Problem: Despite having features and configuration, there are still redundancies and inconsistencies in the variants due to multiple implementations.
L5. Integrated platform with clone and own: The organization integrates the existing variants into a consolidated platform. This is enabled by the metadata for clone traceability and feature-to-asset mappings.
Approach: Using clone traceability, developers consolidate assets belonging to different variants by merging them into holistic assets. The common parts of the implementation are only written once in the consolidated platform, and the variable parts are preserved with feature mappings. The developers can then control the variability by exploiting the feature mappings. Notably, even when a platform is in place, developers can still employ clone&own to their convenience; the recorded clone traceability can be exploited to integrate subsequently cloned variants into the platform if needed.

Problem: Due to cloning, developers still need to ensure consistency by frequently performing change propagation among variants or merging variants into the platform. Having multiple stand-alone variants in addition to the

platform makes it difficult to ensure high quality in both the platform and the variants.

L6. Fully integrated platform: Using metadata, developers merge all variants the organization is maintaining into the consolidated platform. Thenceforth, developers can focus on maintaining and evolving the consolidated platform. This ensures high consistency among the variants, as they are generated from the same platform. Additionally, having a unified implementation facilitates developers to produce high-quality variants.

Approach: Developers merge all variants into the platform, either sequentially, or all at once. To this end, they use the metadata to identify and merge clones, and incorporate variability in the platform when needed.

1.2 Methodology

We discuss the methodology we adopt to answer our research questions in this section.

For **Paper A**, we study frameworks relevant to our work; frameworks supporting variant management or product-line migration. To this end, we gather the frameworks we are familiar with, as well as any frameworks we find from the related work of the papers we know. We find five related frameworks. We evaluate them based on their ability to support activities pertaining to clone management and product-line migration, also analyzing their shortcomings. A summarized table of the comparison of our framework against the other related frameworks is shown in Section 1.4.

We adopt a design-science-like strategy to iteratively synthesize our conceptual structures and operators and evaluate them on realistic scenarios. We create our conceptual structures to reflect assets of different types in the variants' working copy, capturing details relevant for automating tasks such as change propagation and incremental migration etc. With our operators, we aim to maximize the coverage of evolution scenarios we observe from literature as well as our own professional experience. While formulating the conceptual structures, the main challenge is to find a common ground for assets of all types; capturing details which are shared across assets of different types. We strive to find a reasonable trade-off between level of detail and resulting memory consumption, also ensuring that the conceptual structures are language-independent. When designing the operators, the challenge is to ensure that using our framework (*virtual platform*) imposes minimal additional effort. Devising a holistic set of operators which covers different scenarios of software evolution is also a challenge. Another challenge is to formulate the operators in a way that offers high *familiarity* to ensure easy adoption and smooth transition to a software product-line, if needed.

Initial design. The first task is to design the conceptual structures that lie at the foundation of virtual platform. We follow an iterative strategy, conducting multiple brainstorming sessions among the authors. The authors are diversified in terms of experience, two having over ten-year experience in the fields of variability management and SPLE. The discussions involve many small design decisions that play a critical role in the efficiency and effectiveness of our framework. Next, we look into ways to incorporate metadata. The challenge is to find data structures that are light-weight and offer efficient traversal. Lastly, for the operators, we create many ad-hoc scenarios for different types of evolution, each inspired by our experience and informed by real-world projects.

Prototyping and Simulation. We implement our conceptual structures and operators in Scala. After developing each operator, we test it by simulating toy examples that reflect various evolution scenarios. When testing, we check if the operators maintain the validity of the *well-formedness criteria* we define over the conceptual structures. The well-formedness criteria govern the sanity of the conceptual structures, e.g., a file asset cannot contain a folder asset.

Evaluation. Our evaluation is two-fold. For a qualitative assessment, we compare virtual platform with the state-of-art frameworks discussed in Section 1.4. Specifically, we evaluate how the framework supports activities pertaining to variant management or product-line migration. For a quantitative

assessment, we conduct a cost and benefit analysis of using our framework on a real-world system comprising four variants realized via clone&own. To this end, we simulate the development history of a real-world project consisting of four variants. Using Git commands, we retrieve the order of commits developers made during development, as well as the operations performed in each commit (*git diff*). Next, as simulation, we translate each operation performed in the commit to one or more operators in the virtual platform.

During simulation, we count the number of invocations of our operators, each of which has its associated cost. The cost of one operator invocation is the sum of the parameters it has. Ultimately, we have two costs. *Incurred cost* (or added cost) is the accumulated cost of adding feature-related information in the variants. This mainly constitutes adding features in the feature model and adding feature mappings to assets. In the chosen case study, both features and feature mappings were added in textual form in specific files. The *saved cost* is the effort saved due to the elimination of clone detection and feature location. These costs are relevant in scenarios of change propagation in assets, cloning of features, and change propagation in features. These scenarios are studied and documented in a prior contribution [16], where researchers studied the commits and the annotated code to determine the rationale of various evolution activities (e.g., copying assets due to feature cloning across variants). For estimating the saved costs, we make the following assumptions:

- It takes 15 minutes for developers to locate one feature manually (taken from Wang *et al* [37]).
- It takes 15 minutes for developers to find clones of one asset or feature.
- The cost of adding a forgotten mapping is 10 times more than the cost of proactively adding it during development.

We calculate the benefit by subtracting the incurred costs from the saved costs. We discover that we reach a break-even point at 54 seconds; if developers take 54 seconds to add one feature (or mapping), they reach a break-even point. In practice, adding information which is already fresh in the minds of the developers takes significantly lesser time. Additionally, for growing variants with frequent change propagations, the benefit is expected to be incremental.

For **Paper B**, we develop a high-level understanding of the popular process models [11–15] for product-line adoption presented in literature. As explained above, the process models mainly focus on proactive adoption, and assume a strict distinction between domain engineering and application engineering. Additionally, they assume that product lines only evolve through the platform itself; variability is known and implemented at the time of platform construction.

Next, we survey the literature to understand the industrial practices in software product-line adoption. First, each author recommended papers reporting experiences with product-line adoption from their knowledge of the literature. The papers are discussed collectively by all authors, which helps in creating a foundation for our analysis. Second, following the guidelines prescribed by Kitchenham *et al* [38], we manually search for papers published in five conferences (ICSE, ESEC/FSE, ASE, SPLC, VaMoS) and seven journals (EMSE, TSE, TOSEM, JSS, IST, IEEE Software, SPE). We include papers

that are peer-reviewed and written in English, and (optionally propose and) use a process model.

Our final set of papers comprises 32 publications. We classify the articles on two aspects; *product-line adoption*, and *product-line evolution*. Based on our observations, we conclude that there is a gap between the prescribed process models in literature and the deployed industrial practices. Firstly, extractive (12 papers) and reactive adoption strategies (8 papers) are as frequently employed as proactive adoption (12 papers), however, they are not covered by any process model. Secondly, many organizations adopt variant-based product-line evolution (8 papers); evolving platform through variants incrementally. This contrasts the process models, where the assumption is that the entire variant scope is known prior to platform construction, and variability is incorporated in the platform only. Variant-based evolution also negates the assumption that domain engineering and application engineering are executed sequentially; evolved variants (in application engineering) can be merged back into the platform (domain engineering).

Based on these reflections, we conclude that the process models do not depict the industrial practices accurately, and consequently, they should be updated to reflect prevalent practices in product-line adoption and evolution. We present *promote-pl*, an aggregated process model for product-line adoption and evolution. Our process model is less strict than the other process models, focusing on adoption and evolution as the high-level abstraction. In addition to the *proactive* approach, promote-pl also caters for the other approaches for adoption; *reactive* and *extractive*. Next, we discuss our approach for deriving promote-pl.

For each paper, we synthesize a partial order of activities the organization deploying SPLE partook, and unify the terminology to establish a common ground. We compare the partial orders resulting from different reports based on their scope and similarities, and create partitions of our process model. The partitions are along two aspects: adoption and evolution. Finally, we construct our process model by merging all partial orders and removing duplicate activities. The granularity of promote-pl allows practitioners to map the various activities to actual development processes—more flexibly than the existing process models. Additionally, promote-pl is well-aligned with contemporary practices, such as agile software development, clone (or variant) management, incremental product-line adoption, and continuous software engineering.

For **Paper C**, we extend the notion of features in software assets to features in *models* used to analyze and design a system. Specifically, we conduct an experiment comparing different variability mechanisms and their effect on the *comprehensibility* of various model-related tasks. We compare three variability mechanisms: *enumerative*, *annotative*, and *compositional*. We conduct a series of three experiments, each focused on a different model type. The considered model types are class diagrams, state machine diagrams, and activity diagrams. We recruit 164 participants in total (experiment 1: 73 participants, experiment 2: 65 participants, experiment 3: 26 participants), which consist of students from universities in three different countries.

In each experiment, the number of variability mechanisms (n) and subject systems (m) are similar. For each experiment, we model each subject system using all variability mechanisms. Participants are distributed in n groups

randomly. Each participant experiments with each variability mechanism and each subject system exactly once. For experiment 1, we choose three subject systems (d1 = Simulink, d2 = a Project Management System, and d3 = Mobile phone), and three variability mechanisms (Enu = enumerative, Ann = annotative, and Com = compositional). We distribute the participants in three groups. Following the Latin-squared design [25, 26], the groups follow the following paths:

- Enu d1 → Ann d2 → Com d3 (group 1),
- Com d1 → Enu d2 → Ann d3 (group 2), and
- Ann d1 → Com d2 → Enu d3 (group 3).

We only consider annotative and compositional mechanisms in the second and third experiment as the enumerative and annotative mechanisms yield similar results in experiment 1. Consequently, both experiment 2 and 3 feature two subject systems each, and participants are divided into two groups. In experiment 2, we model two high-level features of *Robocode* [39], a programming game employed as a teaching tool. In experiment 3, we model two subject systems: an *Email Service Provider* and a *Flight Reservation System*. As explained in Section 1, for each subject system, we require participants to perform tasks of three types; understanding variants, comparing two variants, and comparing all variants. Each task type consists of two types. For the first task type, the tasks followed the theme: *"Which variants have the elements X and Y? List all such variants, or write none otherwise"*. For the second task type, the tasks followed the theme: *"How do the two variants Var1 and Var2 differ? List all differing elements if there are any"*. For the third task type, the tasks followed the theme: *"Which elements are included in all variants?"*

We measure the time taken by participants by asking them to log the starting and ending time for each subject system. As a post-experiment survey, we ask participants for their subjective preferences for all task types, their preferred mechanism, and the rationale for their choice. Specifically, we ask participants for the following questions, the responses to which are to be given on a 5-point Likert scale (1: very easy, 5: very difficult):

- (S1) How easy did you find it to understand each mechanism?
- (S2) How difficult was it to answer the questions on "Understanding variants" (tasks 1 and 2) for each mechanism?
- (S3) How difficult was it to answer the questions on "Comparing two variants" (tasks 3 and 4) for each mechanism?
- (S4) How difficult was it to answer the questions on "Comparing all variants" (tasks 5 and 6) for each mechanism?

For retrieving the subjective preferences, we formulate the following questions:

- (S5) Which mechanism do you prefer for each of the three task types?
- (S6) Can you explain your subjective preferences (intuitively)?

We evaluate and compare the mechanisms based on their correctness and time taken to perform tasks. As primary evaluation, we assign scores to the participant responses manually. Each task type consists of two tasks, the score of each task can have three values: 0 (incorrect), 0.5 (partially correct), and 1 (correct). A response is only correct if it composes only and all correct elements needed. With three task types having two tasks each, each variability mechanism is scored between 1-6 for each participant. Next, we perform the analysis and comparison. For analyzing our results and plotting the results of our comparison, we use R-scripts, which are shared in the online appendix of Paper C. Our findings indicate that annotative mechanism outperforms the other two mechanisms in two out of three experiments, recognized by better developer performance in terms of accuracy and time taken. The subjective preferences by developers also show a clear inclination towards annotative mechanism in all experiments. We supplement our findings with recommendations; guidelines to support flexible, tailored-to-task solutions, which are summarized in Chapter 4.

1.3 Summary of Contributions

In this section, we summarize the papers this thesis encompasses, focusing on problem statement, aim, solution, and contributions.

1.3.1 Paper A

Customization is a trend in software engineering, and to deliver tailored products to a large target audience, organizations often develop variant-rich systems. To develop variant-rich systems, developers deploy one of two approaches: clone&own and platform-oriented development (a.k.a software product-line engineering). In clone&own, developers reuse existing variants by cloning them and adapting them to satisfy new (or changed) customer requirements. Cloning is an efficient immediate solution, offering ease of adoption and freedom to experiment with features quickly. It is strongly preferred by developers as it does not impose additional training and allows developers to work with (and reuse) artifacts they are familiar with. However, cloning comes with costs. As the number of variants increase, developers lose overview over the variants. The immediate benefits of reuse are soon overcome during evolution and maintenance. To perform simple tasks such as bug-fixes, developers have to find clones of the buggy functionality and replicate the changes in every variant. Introducing new functionality or enhancing existing functionality in all variants is similarly also a difficult task. Due to missing information about clone traceability (what was cloned where), developers have to rely on their memory, or third-party tools for identifying clones. These solutions are far from optimal, as they lead to inconsistencies in the codebase. Another issue is feature location; developers often need to locate functionality corresponding to a certain requirement. Feature location, like clone detection, also relies on human intervention or third-party tools, and leads to inaccurate results.

An alternative way of realizing variant-rich systems is building a fully integrated platform by deploying platform-oriented approaches such as SPLE. A product-line offers better reuse and easier maintenance, as instead of cloning, the focus is on creating *high-quality* core-assets. *Features* are implemented only once (in code assets), and any changes to features (bug-fix, feature enhancement) are also performed in the platform. Developers can configure features to derive new variants corresponding to new customer requirements. Starting a platform from scratch (proactive adoption, Paper B) however, is a tedious task. It relies on tasks that need domain knowledge, as well as incorporation of variability-related concepts (e.g., feature model, variability mechanism, configurator tool etc). SPLE adoption leads to increased time to market, in addition to posing extra costs and resources.

Researchers prescribe a few frameworks [5–7] for migrating variants realized using clone&own to a product line. However, those frameworks, as we will show in Section 1.4 are either too abstract, or rely on heuristics for clone detection and feature location, rendering them inaccurate to be applicable in practice. Most frameworks prescribe a "single-step" migration, which is risky and error prone. Proactive adoption, as we discussed above, can be human-intensive, time-consuming, and risky. Extractive and reactive approaches are limited by accuracy, as they mainly rely on heuristics for clone detection and feature

location. In addition, merging variants in the platform leads to merge conflicts, that require developer intervention to fix.

We aim to bridge the gap between clone&own and SPLE in a minimally invasive, accurate manner. To this end, we devise a framework, virtual platform, that generalizes various variant management and product-line migration frameworks. We propose *truly incremental migration*; migration exploiting explicitly recorded meta-data. We store two kinds of metadata: clone traceability and feature-to-asset mappings. Our framework comprises operators, which mirror the various evolution activities variants go through. The operators work on virtual platform's conceptual structures, which are semi-structured representation of code in memory. The operators serve to keep an in-sync copy of the variant's working copy in memory, as well as recording the metadata mentioned above. A subset of the operators allow developers to query the stored metadata (for example for automated clone detection and feature location). Lastly, virtual platform comprises operators for automated change propagation, feature cloning, and feature-related change propagation. We implement a prototype of virtual platform in Scala and evaluate it on a real, open-source case study comprising four variants. The results indicate the virtual platform saves costs of clone detection and feature location, and provide benefits in terms of automation and accuracy. We present our conceptual structures and operators, as well as the results from our evaluation in Section 1.4.

1.3.2 Paper B

Organizations developing variant-rich systems often need to adopt a product-line approach, either as a result of maintenance overheads, or to satisfy a diverse customer base. Researchers in the recent decades have proposed *process models* [11–15] for product-line adoption. These process models guide practitioners towards product-line adoption by defining concrete activities that need to be performed to build an integrated platform. Typically, the process models comprise two distinct phases: domain engineering (development of core assets) and application engineering (development of individual variants). These product models, although widely established as the de-facto standard for adopting SPLE, focus on the best-case scenario: *proactive* product-line adoption. In reality, SPLE is rarely incorporated from the get-go. Developers start with clone&own, and later migrate to a fully integrated platform when needed. SPLE can be adopted using three approaches: *proactive* (develop a platform from scratch), *reactive* (start with one variant, build incrementally), and *extractive* (start with n variants, migrate in a single step). Additionally, product lines evolve both through the platform itself (as prescribed by the process models), but also through the variants, when developers use contemporary practices to evolve systems naturally.

Realizing that there is a discrepancy between the process models and contemporary industrial practices, we propose an updated process model for product-line adoption. Our process model, promote-pl, is an aggregation of the existing process models as well as the industrial reports for product-line adoption over the recent years. We begin by gathering literature reporting experiences with product-line adoption and evolution in industry. To this end, we manually search 12 venues (including SPLC, ICSE, VaMoS, ASE,

ESEC/FSE, EMSE, and TSE). We limit our search to past five years in order to retrieve only the most recent experience reports. Our inclusion criteria filters papers not written in English and not peer-reviewed. Additionally, we only include papers reporting experiences with *using* a process model, instead of just proposing one. We analyze 32 peer-reviewed articles reporting product-line adoption, and categorize them on two dimensions: adoption and evolution. For each paper, we extract the set of activities undertaken during product-line adoption, and list them in order. Next, we unify the terminology used in papers to reach a common ground. Based on the activities and their partial orders, we synthesize our process model, the high-level representation of which is shown in Section 1.4. We discuss the relevance of our process model in modern software-engineering practices including continuous software engineering, dynamic configurations and adaptive systems, and agile software engineering.

Consisting of multiple entry points (no variant, one variant, many variants), our process model can be used as a guide for product-line adoption and evolution for projects at different stages of maturity. Additionally, promote-pl can be exercised as a teaching tool to teach students the state of the art practices in SPLE in real projects.

1.3.3 Paper C

Organizations employ reuse-oriented development approaches in an attempt to cope with diverse and oft-changing market needs. To prevent the problems stemming from maintaining multiple copies of code, developers incorporate variability in code. To this end, they use variability mechanisms to make the assets configurable, and enable product derivation. To further streamline their development processes, organizations incorporate variability in models as well. This prevents the need to maintain multiple copies of the same model, and simplifies the analysis by having a unified model. Variability mechanisms in models either follow the annotative paradigm or the compositional paradigm. Annotative mechanism allows adding feature annotations on a single, consolidated model. Models corresponding to a specific variant only comprise the model elements annotated with the selected features. Compositional models are fragmented into sub-models (model fragments), each representing one or more features. Models corresponding to a specific variant are a union of the model fragments representing the selected features.

Annotative and compositional mechanisms both have advantages and drawbacks. Annotative mechanism offers a consolidated view, but the models get cluttered when the number of features grows. Compositional models offer a cleaner view, however, they involve the extra cognitive step of "merging" different model fragments to create the final model. At present, the choice of variability mechanism is predominantly done intuitively. Research lacks concrete evidence on how the choice of variability mechanism impacts the efficiency of model-related tasks. We conduct an empirical study to systematically investigate the impact of variability mechanisms on model comprehension tasks for three popular model types: class diagrams, state machine diagrams, and activity diagrams. We conduct three experiments, each focused on one model type. The goal is to guide practitioners towards informed choice of variability

mechanism to use. Another goal is to guide tool developers to build support for flexible, task-tailored solutions.

We recruit 164 students as participants in total for our experiments, where no student performs in more than one experiment. For each experiment, we create models using different variability mechanisms for each subject system. We follow a *Latin square design* [26], such that each participant experiments with every subject system and variability mechanism only once. Participants are required to perform tasks of three types: understanding variants, comparing two variants, and comparing n variants, for each subject system. We ask participants about their subjective preferences after the experiment. Our results indicate that annotative mechanism outperforms the compositional mechanism in two out of three experiments. Additionally, we observe that compositional mechanism impedes developer performance in tasks that require having an overview over the variants. Participants prefer annotative over compositional mechanism in all three experiments. Lastly, participants prefer different mechanisms for different task types, implying that the choice of mechanism depends on the nature of the task. We share a summary of our findings, and our recommendations based on the finding, in Section 1.4.

1.4 Results

In this section, we answer our research questions based on our contributions.

RQ1: *What are the state of the art frameworks for software product line migration and variant synchronization?*
Following, we summarize the five frameworks which informed the design of our framework:
Rubin *et al.* [5, 10] propose an abstract variant-integration framework, specifically a set of operators that a realization of such a framework should realize. The operators are derived from three case studies of organizations migrating cloned variants into a platform. They propose a set of operators that extract feature-oriented information from design documents and code (feature identification), and then link that information to code assets (feature location). They also define operators to mine information about feature dependencies and similarities—all typically require complex algorithms. After extracting this data, they reorganize the variants into clusters of related assets. Lastly, they compose various assets that result in a tentative architecture and a feature model. The application of the framework is discussed for three case studies.

Notably, the framework supports the narrative that an operator-based perspective leads to more efficient implementation and support. However, while their framework is abstract in nature; our method and tool can be seen as a realization, relying on recording and exploiting metadata. Consider their operator `findFeatures`, which is described as an operator that identifies features from assets, which usually requires expensive and inaccurate feature-location effort. In our case, we can easily retrieve this information from the *asset tree* (explained shortly). Furthermore, their work emphasizes the need to support different types of assets (code, requirements, design models, and tests), which we support with a programming-language-independent representation.
ECCO is a variant integration framework and tool, developed by Fischer *et al.* [6], for composing new product variants using reusable assets. The input to the framework is a set of variants realized via clone&own, each with a list of implemented features; additional cloned variants may be added over time. *Extraction* is used to collect information about features and assets, as well as the feature interactions. *Commonality analysis* on variants leads to a set of reusable assets along with dependencies between them. *Composition* takes a selection of features and uses the reusable assets from 'extraction' to compose a partial or complete product. *Completion* guides the developers to fill in the missing implementation corresponding to features and feature interactions. The new variant is fed back into the system. While the framework significantly improves reuse, the accuracy of the framework is debatable, since it primarily relies of heuristics for feature identification, feature location and clone detection. In comparison to ECCO, our framework is more accurate, since it does not rely on recovering metadata, but instead on actively logging and maintaining it. Additionally, as opposed to ECCO, we support change propagation for synchronizing variants after cloning.
BUT4Reuse, presented by Martinez *et al.* [7], is an extraction-based technique for variant integration. The process begins with the creation of an asset model using the asset variants. Then, software product-line adopters perform feature

identification and feature location in the variants. Using the resultant list of features and the constraints among them, a feature model is automatically created. Lastly, using feature location traces, the reusable assets are created by extracting implementations of all features, and made suitable to be included in a variant by composition. Since the goal is to facilitate a "one-shot" migration of a large set of variants towards a platform, feature identification and location need to be performed exhaustively. Due to the lack of reliable automated techniques for the tasks, BUT4Reuse heavily relies on the involvement of domain experts for these tasks. These tasks are daunting and error-prone, rendering the framework risky in practice. Additionally, this requires some training for technical tasks as majority of these activities are different than what developers do in real-time. Our framework offers a higher degree of familiarity, with the only added effort being the addition of metadata.

VariantSync, developed by Pfofe *et al.* [8], is a plugin-supported framework for automated synchronization of related software variants. The plugin uses FeatureIDE [40] for feature model specification, and allows developers to tag code fragments to feature expressions. After a change, developers can choose to perform a source- or target-focused change propagation. The former propagates the changes to all fragments whose tagged feature expression evaluates to true against the changed code's feature expression, whereas the latter only propagates changes to a specified code fragment. The merge conflicts arising during change propagation are resolved manually using java-diff-utils[§§]. The framework is light-weight and eliminates feature location and clone detection. It also saves maintenance effort as the change propagation is automated. However, it is unclear how an asset can be reused automatically when a feature expression is tagged with it. Moreover, VariantSync only considers code fragments, but in reality, features could contain implementations in entire directories and files, as is supported in our framework.

Montalvillo and Díaz [9] propose supplementing version control systems (VCS) with operations that allow synchronization between artifacts belonging to two distinct phases of product line development; Domain Engineering (DE) and Application Engineering (DE) [11]. To this end, they specify a repository architecture comprising of a *core asset repository* and a set of *product repositories*. On top of these, they define branching models that allow isolated development of core assets and customized products with two-way propagations when needed. *Update propagations* allow disseminating the changes in core assets to product repositories, while *feedback propagations* feed changes introduced in product repositories back into the core assets repositories. Traditional VCS operations are used to provide these propagations, where development takes place in branches (*branch, fork*) and product repositories use (*clone*) assets from the core assets repository. Changes made to core assets or product specific assets are propagated (*merge*) when required. To automate their approach, they use web-augmentation techniques to enhance GIT with the proposed operations. The approach is novel and flexible, however it comes with the risk of frequent need of resolving merge conflicts. Also, for a large number of products, the storage requirements inevitably grow and it can be difficult to keep track of which changes to propagate and which branches to propagate to. Lastly, the need to locate features will eventually rise and without

[§§] `https://github.com/KengoTODA/java-diff-utils`

Table 1.1: Comparison of the virtual platform with activities supported by clone management and product-line migration frameworks

Feature identification → abstract operator [10], specified in the beginning [6–8], specified any time in virtual platform
Feature location → abstract operator [10], extracted [6, 7], internal tagging [8], also internal tagging in virtual platform
Feature dependency management → abstract operator [10], statically mined [7], specified in beginning [8], specified any time in virtual platform
Feature model creation → multiple abstract operators [10], activity [7], specified in the beginning [8], dynamically grows in virtual platform
Feature-to-asset mapping → abstract operator [10], extracted [6, 7], specified any time [8], specified any time in virtual platform
Clone detection → textual diff tools [10], feature expression comparison [8], git clone points to source [9], not needed in virtual platform
Feature cloning → supported by virtual platform
Change propagation → multiple abstract operators [10], variant synchronization [8], using Git merge [9], automated in virtual platform
Reusable assets creation → abstract & incremental [10], reuse existing variants [6], reusable core assets [7, 9] and features in virtual platform
Product derivation → abstract [10], customizing after cherry-picking [9], composition [6, 7], preprocessor-like in virtual platform
Integration → abstract operator using meta-data [10], third party tool [8], Git merge [9], manual or tool-based, guided by meta-data in the virtual platform
Variant synchronization → Git diff [9], code comparison [6, 7], not needed in virtual platform

a notion of features, it can be difficult to trace which changes in implementation corresponding to a certain feature.

Table 1.1 shows a comparison of the frameworks analyzed in RQ1 against each other and virtual platform. Contrary to other frameworks, virtual platform does not rely on heuristics or third-party tools for extracting or specifying features, as well as their locations in assets. It also allows developers to flexibly add feature dependencies at any time during development. Additionally, virtual platform allows developers to dynamically grow and shrink feature models in parallel to software evolution instead of recovering them heuristically or specifying them in the beginning. Only two frameworks, VariantSync [8] and virtual platform offer accurately adding feature mappings in assets, however, virtual platform allows mapping features to not only code, but assets of different types (e.g., repository, folder, file etc). Virtual platform is the only framework that supports feature cloning: cloning all assets implementing a feature, to another variant. It also eliminates the need for clone detection and feature location, allowing developers to query the stored metadata to retrieve such information automatically. The stored metadata also enables automated change propagation, along both assets and features. Assets and features can undergo changes after cloning (e.g., renaming). Using accurately stored metadata, changes can be propagated among clones, ensuring a consistent implementation across variants. The metadata can also be exploited for two-way change

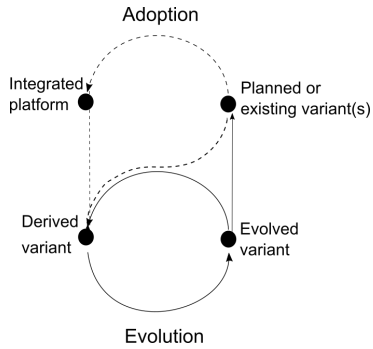


Figure 1.4: High-level overview of promote-pl from Paper B

propagation (variant synchronization), variant integration (combining multiple variants into one), and automated product derivation.

RQ2: *What are the contemporary product-line engineering processes employed in practice? Do they accurately depict the process models for product-line adoption presented in literature?*
As explained above, we survey 32 papers reporting experiences with product-line adoption and evolution. Confirming the observations from our experience with industrial collaborations, a considerable proportion of the studied papers report extractive (12 instances) and reactive (8 instances) adoption in addition to the standard approach assumed by the process models; proactive adoption (12 instances). Additionally, we observed that product lines also evolve through their variants (7 papers reporting variant-based evolution) instead of just evolving through their platforms. Recognizing these discrepancies, we conclude that the process models need to be updated to reflect realistic industrial practices. To this end, we synthesize a common process model, the methodology for which is described earlier in Section 1.2. Figure 1.4 represents a high level abstraction of our process model (for the larger version, see Chapter 3).

Developers can start with a platform from scratch, leading them to start with the "integrated platform" in Figure 1.4 (proactive adoption). They can also start with one or more variants (top-right corner), and migrate to the platform incrementally (reactive adoption) , or all at once (extractive adoption). For creating new variants (derived variant), developers can follow one of two routes. They can derive a variant from the integrated platform by providing a valid feature selection. Alternatively, they can reuse an existing variant (clone&own) to create a new variant. The copied variant can be evolved (evolution) to create the new, evolved variant. Lastly, if developers want to merge the derived and evolved variants back into the platform, they can switch back to adoption, merging variants sequentially or simultaneously into the platform.

Notably, our process model:

- switches the focus from the primary decomposition between domain engineering and application engineering to adoption and evolution.
- is round-trip—allowing practitioners to switch between adoption and

evolution conveniently.

- covers all adoption strategies, allowing developers to approach product-line adoption in different ways.
- caters for both platform-based evolution and variant-based evolution, as opposed to only assuming that product lines evolve through their platforms (as done by the process models).
- is based on concrete evidence and experience from industrial collaborations with different companies.
- comprises activities at various granularity levels, allowing developers to easily map and apply the activities to various development processes, less strictly than the existing process models.
- connects software product-line engineering to contemporary industrial practices including agile practices, clone management, incremental software product-line adoption, dynamic configuration, and continuous software engineering.

RQ3: *How to design an appropriate framework to realize the truly incremental migration? Specifically, what operators and conceptual structures need to be defined, and how?*
As explained above, we devise a framework for truly incremental migration, starting from a lean set of variants to a fully integrated platform. Our framework, virtual platform, relies on accurately stored metadata for activities such as change propagation and software product-line migration. Next we elaborate the conceptual structures and operators that formalize our framework.

Conceptual structures. Our conceptual structures store semi-structured representations of assets at different levels of granularity, as well as the features they map to. Figure 1.5 shows a demonstration of our conceptual structures. `Asset` represents an artifact pertaining to a variant. Assets can have a `name` and a numeric `version` number (versioning explained shortly). Additionally, assets can have an `AssetType` depending on their granularity level in the variant. An `Asset` can be one of the defined asset types: root (`VPRootType`, virtual top node to hold all variants), repository (`RepositoryType`), folder

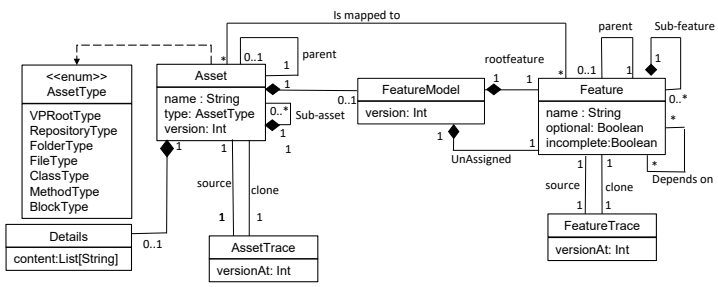


Figure 1.5: Conceptual structures for the virtual platform

(FolderType), file (FileType), class (ClassType), method (MethodType), and block (BlockType). Assets are hierarchical; an Asset can have sub-assets (e.g., folders can have files), making each Asset an Asset Tree (AT). Each Asset, except the root asset, is parented (e.g., class is a parent of method). Textual assets also have Details, which store the content the textual files comprise. Each Asset optionally can also have a Feature Model (explained shortly). For flexibility, we allow developers to add a Feature Model in all types of assets. Additionally, for flexibility, we also allow multiple feature models in a variant; the feature models lower in the AT will have more fine-grained features. Once a Feature Model is added to an Asset A, the assets in the hierarchy of A can be mapped to any feature from the FM. Assets are mapped to features (using presence conditions). Each Asset has a presenceCondition, which is an expression over the features it maps to (e.g., *Messaging* & *Internet*). The default value of presenceCondition is True; features are mapped to the Asset by adding them as a disjunction to the presenceCondition. The rationale for adding with a disjunction is to make the mapping flexible and not too conservative, such that we can say that the Asset is included in the final configuration if any (or both or all) of the features is selected. Feature, like an Asset, is named. Additionally, it has a boolean parameter optional to represent if it is a mandatory or optional feature. Features can also have dependencies (Depends On). Lastly, the parameter incomplete represents if all assets mapped to the feature are in the variant or not. Features are contained in a Feature Model, which is a hierarchical structure composed of features. In a Feature Model, features have parent features and sub-features (similar to assets). Features, like assets, are versioned. Feature models also have an Unassigned feature, which is used to add features whose target (or parent) features have not been provided. Feature models also hold a reference to the assets they belong to. This enables tracking which Asset the Feature Model belongs to. Lastly, for storing clone traces, there are two trace databases; an AssetTraceDatabase and a FeatureTraceDatabase. The trace databases are simplistic. Each database comprises traces; each trace storing a link to the source Asset (or feature), its clone, and the version of source at the time the Asset was cloned. The version is used to determine if there are changes to be propagated in change propagation.

Versioning. Aiming for simplicity, we use numeric versions, starting from 0. An Asset not yet added in the asset tree (AT) has a version of 0. The version of the root node is called the globalVersion, and it has special relevance in the versioning strategy. After each update in the AT, the globalVersion is incremented and assigned to the updated assets. Our operators update the versions of the assets. The same protocol is applied to features in the Feature Model. Such a strategy saves expensive tree traversal; only updating the affected assets after an operation has been performed.

Asset-oriented operators. Asset-oriented operators depict routine tasks developers perform. These conventional operators serve two purposes. First, they maintain an in-sync copy of the working copy of the variants in the AT. Second, they store metadata pertaining to clones and features. The operators also govern versioning of the various conceptual structures explained above.

Table 1.2: Summary of asset-oriented operators

Operator	Summary
`AddAsset(S,T)`	Adds a given asset S to a given parent asset T (in the virtual platform's AT). Updates the `globalVersion` and assigns it to S and T. Also adds any features mapped to S in the feature model of T's repository.
`ChangeAsset(S)`	Changes a given asset S (e.g., renaming, addition of a line etc) in the virtual platform's AT. Updates the `globalVersion` and assigns it to S.
`RemoveAsset(S)`	Removes an asset S from the AT. Updates the `globalVersion` and assigns it to the parent of S before deletion. Also removes any features *only* mapped to S from the repository's feature model.
`MoveAsset(S,T)`	Moves an asset S from its parent asset to another asset T. Is a logical combination of `CloneAsset` and `RemoveAsset`. No traceability is stored in this operator.
`CloneAsset(S,T)`	Clones an asset S from one variant (or repository) to an asset T belonging to another variant. Also clones any features S maps to into T's repository's feature model. The clones retain the version of the source assets. The `globalVersion` is incremented and assigned to T. Cloning is deep; the operator creates a deep clone of the asset down to the leaf nodes and adds it to the target variant. A new trace is created, pointing to the source asset(s), its clone, and the version of the source when it was cloned. The trace is added to the `AssetTraceDatabase`.
`PropagateTo -Asset(S,S*)`	Propagates changes in an asset S to its clone S*. Traverses the `AssetTraceDatabase` to determine if the assets have a source-clone relationship. If they do, determines if a change propagation is valid by comparing the current version of S to the one at the time of cloning. Propagates changes automatically if both the conditions are true. Changes that can be propagated are renaming, addition of a sub-asset, mapping to a new feature, and change in the content (in case of textual assets) etc. Propagation is also deep; change propagation in an asset is executed in all its sub-assets down to the leaf node.

Table 1.2 presents a summary of our asset-oriented operators, focusing on the functionality and versioning protocol of each. The operators for maintaining an in-sync copy of the AT in the memory are `AddAsset`, `ChangeAsset`, `RemoveAsset`, and `MoveAsset`. The operators that add metadata pertaining to clone traceability are `CloneAsset` and `PropagateToAsset`.

Notably, asset-oriented operators are mostly invoked silently in the background (except `PropagateToAsset`, which needs to be invoked explicitly). The metadata for clone traceability is also automatically stored and queried (by convenience operators, explained shortly).

Table 1.3: Summary of feature-oriented operators

`AddFeatureModelToAsset(A,FM)` → Adds a given feature model FM to an asset A of any type. Also adds a reference to A in the FM. Updates the `globalVersion` of the asset tree A belongs to, and assigns it to A.
`AddFeatureToFeatureModel(F,T,FM)` → Adds a feature F to either a given target feature F, or to a feature model FM. If T is not provided, F is added as a top-level feature of the FM. The `globalVersion` of the FM is incremented and assigned to S (and T if provided).
`MapAssetToFeature(A,F)` → Checks if the given feature F belongs to the feature model closest to A in its asset tree, and if it does, maps A to F by adding F to the `presenceCondition` of A with a disjunction. Updates the `globalVersion` of the AT and assigns it to A. If the feature does not exist in the feature model of A's closest ancestor with a feature model, it is added to the `Unassigned` feature of the feature model before mapping.
`ChangeFeature(F)` → Changes a feature F in the feature model. Examples of changes are renaming and addition of a feature dependency. Updates the `globalVersion` of the FM and assigns it to F.
`MakeFeatureOptional(F)` → Makes a feature optional. Virtual platform allows developers to quickly and conveniently govern configurations and provide adaptability by making features optional. The `globalVersion` of the FM is incremented and assigned to F.
`RemoveFeature(F)` → Removes a feature F from the feature model it belongs to. Also removes assets from the relevant AT that only map to F. Updates the `globalVersion` of the FM and assigns it to F's parent before feature removal.
`MoveFeature(F)` → Moves a feature from one parent feature to another in the same feature model. Is a logical combination of `AddFeatureToFeatureModel` and `RemoveFeature`.
`CloneFeature(F, TF, TFM)` → Clones a feature F from a source feature model SFM to a given target feature (TF) or a target feature model (TFM). Also clones any assets from the source AT mapped to F in the target's AT. For cloning the mapped assets, it uses *tree slicing* to get a slice of the source AT relevant to the mapped asset. The feature clone and the (mapped) asset clones all retain their source versions. The `globalVersion` of TFM is incremented and assigned to TF. Feature cloning is deep; the feature is cloned along all its sub-features down to the leaf nodes. The mapped asset cloning is also deep. After cloning, traces for the feature and asset clones are added in the `FeatureTraceDatabase` and `AssetTraceDatabase` respectively.
`PropagateToFeature(F,F*)` → Propagates changes in a feature F to its clone F*. Traverses the `FeatureTraceDatabase` to determine if the features have a source-clone relationship. If they do, determines if a change propagation is valid by comparing the current version of source feature to the one at the time of cloning. Propagates changes automatically if both the conditions are true. Changes that can be propagated are renaming, addition of a sub-feature, and mapping to a new asset etc. Change propagation is deep; propagation in a feature recursively invokes propagation in all the sub-features. After propagation, a new trace between the feature and its clone with the current version of the source feature is created and added to the `FeatureTraceDatabase`.

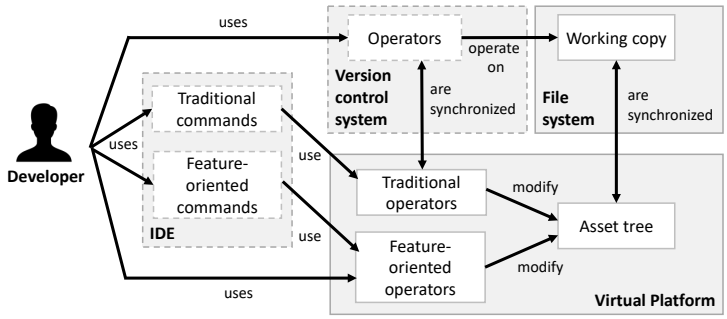


Figure 1.6: High-level overview of virtual platform

Feature-oriented operators. Feature-oriented operators incorporate feature-related data into software assets mainly by allowing developers to add a `Feature Model`, update it (add, remove, and change features), and add mappings between features and assets. As explained above, virtual platform allows developers to add feature models in assets of all types. Additionally, feature models can grow and shrink dynamically, any time during the development. Developers can also make features optional or mandatory, and add dependencies between them. Virtual platform also offers systematic feature reuse; developers can clone entire features across variants. To clone a feature, developers simply invoke `CloneFeature`, which automatically retrieves all assets implementing the feature, and clones them to the provided target variant. Features can evolve independently once cloned, and synchronized using `PropagateToFeature`. Table 1.3 presents a summary of our feature-oriented operators, focusing on the functionality and versioning protocol of each.

Convenience operators. Convenience operators are the helper methods that are used to query the various conceptual structures and retrieve information needed for performing other tasks (such as feature cloning or change propagation). For brevity, we omit the details of the convenience operators. A detailed description of each convenience operator can be found in the online appendix of Chapter 2. Some examples of convenience operators are `getMappedAssets` (retrieve assets implementing a given feature), `getClones` (get clones of a given `Asset` or `feature`), and `getLatestTrace` (get the last trace between a given `Asset` or `feature` and its clone) etc.

Virtual platform overview. Figure 1.6 represents the various way developers can incorporate virtual platform. Virtual platform offers both direct and indirect interaction. Developers can directly invoke the operators using a command-line interface (or a graphical user interface). Notably, asset-oriented operators (except explicit change propagation) do not require manual invocation. An indirect interaction can also be enabled by provided hooks and extensions in existing development tools or version-control systems; addition of a file in a version-control system can be linked to the `AddAsset` operator in virtual platform.

Next, we discuss the details of the evaluation.

RQ4: *What is the effectiveness of the framework virtual platform? Specifically, what are the costs and benefits?*
We choose Clafer Web Tools for our analysis. Clafer Web Tools (CWT) is an open-source system having four variants, three of which are clones (ClaferIDE, ClaferMooVisualizer, and ClaferConfigurator), and one is an integrated platform (ClaferUICommonPlatform). The development of these systems involved significant cloning between the projects. We follow the dataset by Ji et al. [16], who incorporated feature-related data in the sub-systems, as if they were implemented in a feature-oriented way. Features were incorporated in two ways; in the `Feature Model`, and in mappings. Features were mapped to different types of assets including directories, files, and text (e.g., code).

As explained above, we retrofit our operators to the activities performed during the simulation performed by Ji et al. [16]. For each two consecutive commits, we retrieve the summary of tasks performed by developers using *git diff*, and translate the changes in the commit into virtual platform operators. Virtual platform automatically creates an AT, comprising repository assets for all variants. The operators synchronize the AT with the working copies of all variants by invoking operators equivalent to actual evolution tasks developers perform.

Next, we measure the cost of using virtual platform. Notably, our asset-oriented operators do not impose any additional costs since they mirror actual developer tasks, and are automatically invoked in the background. The only added costs are those of adding and maintaining features (C_{feat}), and the cost of dealing with missing features that the developers forgot to add (C_{miss}). For the former, we count the number of times a `feature` was added (# of invocations of `AddFeature`) and the number of times a `feature` was mapped to an `Asset` (# of invocations of `MapAssetToFeature`). Table 1.4 shows the number of invocations for all operators. There are 724 invocations of feature-oriented operators, the most commonly invoked ones being `AddFeature` (368) and `MapAssetToFeature` (229). The actual cost of invoking these operators is assumed to be low (in seconds), as features exist in the developers' mind when coding, and it takes little effort to materialize them for recording. For measuring C_{miss}, we count the number of *late* invocations of `MapAssetToFeature`;

Table 1.4: 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

situations where developers realized a `feature` annotation was missing, and retroactively added it. There are two operators that rely on explicitly recorded feature-related metadata; `CloneFeature` and `PropagateToFeature`. If features are not recorded, each operator needs extra manual effort for feature location. We therefore count the number of times a `feature` mapping was added (`MapAssetToFeature`) before or after the above-mentioned operators (`CloneFeature` and `PropagateToFeature`) were invoked. Additionally, we also look for the number of times a `feature` itself was added (`AddFeature`) before or after the operators `CloneFeature` and `PropagateToFeature` were invoked. We find 14 invocations of `MapAssetToFeature` relevant to `CloneFeature`, and none relevant to `PropagateToFeature`. For `AddFeature`, we find 25 invocations, yielding 39 invocations in total for C_{miss}.

Last, we measure the saved costs. To this end, we account for the times virtual platform bypasses manual clone detection (C_{clone}) and feature location (C_{loc}) due to explicitly recorded metadata. For measuring C_{clone}, we count the number of times developers did not need to locate clones of assets for propagating changes in them (`PropagateToAsset`). We find 8 such invocations. To count the actual cost, we multiply the number of invocations with the amount of time developers take to manually perform clone detection (15 minutes as explained in Section 1.2). For measuring C_{loc}, we count operators that require feature location, which is done automatically by virtual platform (resulting in cost savings). The operators `CloneFeature` and `PropagateToFeature` rely on explicitly recorded `feature` mappings. We assume that each invocation of these operators results in saved cost in terms of time spent in locating one `feature` (15 minutes as explained in Section 1.2). We count 54 invocations of `CloneFeature` and 7 relevant invocations of `PropagateToFeature`, leading to a total value of 61.

Break-even point. We conduct a break-even point analysis since in our evaluation, virtual platform is not used alongside development, but rather to simulate development afterwards. We calculate the benefit by using the following formula:

$$B_{total} = \textit{saved costs - added costs}$$
$$B_{total} = (C_{clone} + C_{loc}) - (C_{feat} + C_{miss})$$

Based on our analysis, if developers take **54 seconds** to record one `feature` (or feature mapping), they reach a break-even point; when the saved costs and added costs even each other out. In reality, developers take significantly less than 54 seconds to add (and map) features, as they are familiar with the features they are developing. The only effort required is the cognitive process of choosing a meaningful `feature` name when adding a feature. In case of `feature` mappings, given that the features are already known (and added in the `Feature Model`) and the notation for specifying `feature` mappings is established, the effort is expected to be even less than that of adding features. We envision higher accuracy if virtual platform is used in parallel to development. Additionally, for larger-scale systems with many features and frequent cloning, the benefits are expected to be exponential.

RQ5: *Can features be leveraged to facilitate model comprehension?*

Table 1.5: Correctness scores for experiment 1 (class diagrams).

	Annotative			Compositional			Enumerative		
Task type	**Mean**	**Mdn**	**Sdt.dev**	**Mean**	**Mdn**	**Std.ev**	**Mean**	**Mdn**	**St.dev**
1: Tracing elements to variants	1.7/2	2.0/2	0.6	1.5/2	2.0/2	0.7	1.7/2	2.0/2	0.6
2: Comparing two variants	1.5/2	1.5/2	0.6	1.5/2	1.5/2	0.6	1.6/2	1.5/2	0.4
3: Comparing all variants	1.6/2	2.0/2	0.7	1.2/2	1.0/2	0.7	1.5/2	2.0/2	0.7
Total	4.8/6	5.0/6	1.3	4.2/6	4.5/6	1.4	4.9/6	5.0/6	1.1

Mdn: Median, **St.dev:** Standard deviation

Table 1.6: Correctness scores for experiment 2 (state machine diagrams).

	Annotative			Compositional			Enumerative		
Task type	**Mean**	**Mdn**	**Sdt.dev**	**Mean**	**Mdn**	**Std.ev**	**Mean**	**Mdn**	**St.dev**
1: Tracing elements to variants	1.2/2	1.0/2	0.8	1.2/2	1.0/2	0.7	-	-	-
2: Comparing two variants	1.0/2	1.0/2	0.8	1.1/2	1.0/2	0.8	-	-	-
3: Comparing all variants	1.2/2	1.0/2	0.8	0.8/2	1.0/2	0.7	-	-	-
Total	3.4/6	3.0/6	2.0	3.1/6	3.0/6	1.8	-	-	-

Mdn: Median, **St.dev:** Standard deviation

Table 1.7: Correctness scores for experiment 3 (activity diagrams).

	Annotative			Compositional			Enumerative		
Task type	**Mean**	**Mdn**	**Sdt.dev**	**Mean**	**Mdn**	**St.dev**	**Mean**	**Mdn**	**St.dev**
1: Tracing elements to variants	1.4/2	1.5/2	0.6	1.2/2	1.0/2	0.6	-	-	-
2: Comparing two variants	1.4/2	1.5/2	0.6	0.9/2	1.0/2	0.7	-	-	-
3: Comparing all variants	1.4/2	1.5/2	0.5	0.9/2	1.0/2	0.6	-	-	-
Total	4.2/6	4.0/6	1.0	3.0/6	3.0/6	1.3	-	-	-

Mdn: Median, **St.dev:** Standard deviation

Following, we share the results of our analysis from the three experiments. For brevity and conciseness, we provide a high-level overview of the results in this report. For a detailed insight into the results of hypothesis testing and effect-size analysis [41], please refer to Chapter 4. Table 1.5 - 1.7 represent the correctness scores for the three types of tasks using each featured variability mechanism in each experiment. For deeper insight, in addition to the average scores, we represent the median (Mdn) scores and standard deviation (Std. dev) observed in the scores as well. We discuss the results on two dimensions: correctness and time taken.

Correctness. In experiment 1 (Table 1.5), annotative and enumerative mechanisms lead to similar correctness scores (no significant differences), whereas compositional mechanism leads to lowest scores in all task types. Hypothesis testing reveals a significant difference between compositional and other two mechanisms for task type 1 and 3. In experiment 2 (Table 1.6), for task type 1 and 2, there are no significant differences between the correctness scores using annotative and compositional mechanisms. We observe a significant difference for task type 3, the most complex task type. A possible rationale for the lower score in task type 3 using compositional mechanism is that while the other task types can be answered without an in-depth understanding of how composition works, task type 3 requires deeper knowledge. Using compositional mechanism requires the extra cognitive step of composing various model fragments to form the complete model, which can be challenging and inefficient. In experiment 3 (Table 1.7), we find more notable differences. Participants using annotative

Table 1.8: Completion times (in minutes) of our participants for all three experiments.

Exp	Mechanism	Min	Mn	Mdn	Max	St.dev
1	**Annotative**	3	6.6	6	15	2.6
	Compositional	4	8.8	8	17	3.2
	Enumerative	3	7.1	6	19	3.1
2	**Annotative**	3	10.4	10	23	4.3
	Compositional	5	11.3	11	22	3.7
3	**Annotative**	3	14.2	14	28	6.1
	Compositional	8	16.8	14	32	6.7

Mn: Mean, **Mdn:** Median, **St.dev:** Standard deviation

mechanism outperform the others for all task types. Hypothesis testing also reveals significant differences for all task types. On average, the correctness score for participants using annotative mechanism is 1.3 times higher than the correctness score using compositional mechanism.

Completion Time. Table 1.8 represents an overview of the times taken using each variability mechanism in all three experiments. For each experiment, we show the minimum (Min) and maximum (Max) time taken, the average time (Mn), the median value of completion time (Mdn), and the observed standard deviation (Std. dev). Participants are fastest in **Experiment 1** (see median times), possibly due to the tasks being simpler for all task types. Participants using annotative mechanism solve the tasks slightly faster than the ones using enumerative mechanism, the difference is however not significant. Compositional mechanism leads to the slowest completion times, with significant differences in comparison to both annotative and enumerative mechanisms. For **Experiment 2**, participants spend almost equal time using both mechanisms, with no significant differences revealed in hypothesis testing. This is consistent with the correctness scores, where we find no statistical differences for the first two task types. For **Experiment 3**, participants using annotative mechanism are 2.6 minutes faster on average than participants using the compositional mechanism. Hypothesis testing reveals that the difference is not significant. In summary however, annotative mechanism leads to the fastest average completion times in all three experiments.

Subjective perceptions. We ask participants about the ease of understanding each mechanism, and the difficulty they experience with each task type using all variability mechanisms. Table 1.9 shows the participant ratings for understandability and difficulty using all variability mechanisms in each experiment. In **Experiment 1**, enumerative mechanism is rated the easiest to understand (2.2 mean rating), followed by annotative (2.6) and compositional (3.2) mechanisms. The analysis of understandability ratings between all mechanisms leads to significant differences. Participants experience similar difficulty using both annotative and enumerative mechanisms; with no statistical differences identified in hypothesis testing. Compositional mechanism is rated the most difficult in all task types. The analysis for all comparisons with compositional mechanism reveal significant differences. In **Experiment 2**,

Table 1.9: Participant perceptions (understandability and difficulty ratings) for Experiment 1, 2, and 3.

			Annotative			Compositional			Enumerative		
Exp	**Quality**		**Mean**	**Mdn**	**St.dev**	**Mean**	**Mdn**	**St.dev**	**Mean**	**Mdn**	**St.dev**
1	**Understandability**		2.6/5	3/5	1.1	3.2/5	3/5	1.1	2.2/5	2/5	1.2
	Difficulty	Task type 1	2.3/5	2/5	1.2	3.1/5	3/5	1.2	2.3/5	2/5	1.1
		Task type 2	2.5/5	2/5	1.3	3.0/5	3/5	1.3	2.2/5	2/5	1.2
		Task type 3	2.5/5	2/5	1.2	3.2/5	3/5	1.3	2.5/5	2/5	1.1
2	**Understandability**		3.0/5	3/5	1.0	2.9/5	3/5	0.9	-	-	-
	Difficulty	Task type 1	2.6/5	3/5	1.0	2.6/5	3/5	1.0	-	-	-
		Task type 2	2.6/5	3/5	0.9	2.6/5	2/5	0.9	-	-	-
		Task type 3	2.9/5	3/5	1.0	2.9/5	3/5	1.0	-	-	-
3	**Understandability**		2.7/5	2/5	1.3	3.4/5	3/5	1.1	-	-	-
	Difficulty	Task type 1	2.3/5	2/5	1.0	3.2/5	3/5	0.9	-	-	-
		Task type 2	2.5/5	2/5	1.3	3.0/5	3/5	1.3	-	-	-
		Task type 3	2.5/5	2/5	1.2	3.2/5	3/5	1.3	-	-	-

[1] Scores on a 5-point Likert scale with 1: very easy, 5: very hard to understand.
[2] Scores on a 5-point Likert scale with 1: very easy, 5: very difficult to perform task.
Mdn: Median, **St.dev:** Standard deviation

in-line with the observations for correctness and completion times, participants experience similar level of difficulty using both annotative and compositional mechanisms. The understandability ratings are also similar. No significant differences are found in hypothesis testing. In **Experiment 3**, annotative mechanism is rated to be more understandable (2.7 vs 3.4) and less difficult to deal with (2.7, 2.3, 2.5 vs. 3.2, 3.0, 3.2) for all three task types in comparison to the compositional mechanism. The analysis of understandability ratings and the difficulty ratings for task type 2 reveals significant differences in hypothesis testing.

Table 1.10: Distribution of preferred mechanisms per task type

Task type	**Ann.**	**Com.**	**Enu.**	**None**
Experiment 1				
1 Understanding variants	50.7%	13.7%	34.2%	1.4%
2 Comparing two variants	26.0%	15.1%	57.5%	1.4%
3 Comparing all variants	43.8%	12.3%	42.5%	1.4%
Experiment 2				
1 Understanding variants	58.6%	33.8%	-	7.6%
2 Comparing two variants	52.3%	41.5%	-	6.2%
3 Comparing all variants	46.2%	41.5%	-	12.3%
Experiment 3				
1 Understanding variants	78.3%	8.7%	-	13.0%
2 Comparing two variants	78.3%	21.7%	-	0%
3 Comparing all variants	78.3%	17.4%	-	4.3%

Ann: Annotative Variability **Com:** Compositional Variability **Enu:** Enumerative Variability

Participant Preferences. As explained above, we ask participants for their preferred mechanism for each task type (S5, S6, Section 1.2). Table 1.10 shows the distribution of participant preferences for all task types. For **Experiment 1**, we find notable differences in the participant preferences. Majority of

the participants prefer annotative mechanism for task type 1 (50.7%) and 3 (43%). Enumerative mechanism is favored by participants for task type 2 (57.5%), possibly because it shows a side-by-side view of variants, making it easier to compare them. Compositional mechanism is the least preferred mechanism (13.7%, 15.1%, 12.3%). For **Experiment 2**, participants clearly prefer annotative mechanism (58.6%, 52.3%, 46.2%) over the compositional mechanism (33.8%, 41.5%, 41.5%). The differences are more pronounced for task type 1 and 2, possibly because the consolidated view facilitates the reader to understand and compare variants. For **Experiment 3**, the differences are more notable. Majority of the participants prefer annotative mechanism (78.43%, 78.43%, 78.43%) over the compositional mechanism (8.7%, 21.7%, 17.4%). These percentages align well with the correctness scores and completion times, where participants perform better and faster for all task types in Experiment 3.
Qualitative Responses. After asking participants to describe their intuition for preferring their chosen mechanism in S5, we manually assess the responses and use *inductive coding* to tag the participants' comments. We observe that participants prefer a mechanism based on the conciseness it offers, the ease of understandability, the degree of familiarity, scalability, and efficiency. Participants also find labels and colors helpful. For behavioral diagrams (state machine diagrams and activity diagrams), participants prefer mechanisms that offer a better flow (annotative and enumerative). As recommendations, we suggest practitioners to provide flexible, task-oriented solutions. Additionally, for smaller systems with fewer features, we advise using the simplest representation: the enumerative mechanism. We also urge developers to use labels and colors to improve readability of the models and assist developers in their routine tasks.

1.5 Conclusion and Future Work

In this work, we present virtual platform—a framework for incremental migration of variants realized using clone&own into a fully integrated platform using explicitly recorded metadata. We determine that software product-line engineering is not always adopted proactively (Paper B). Instead, organizations often choose to transition to an SPL-based architecture only when they face reuse and maintenance overheads as a result of using clone&own. Consequently, they migrate to an SPL-based architecture either reactively or extractively. Researchers have presented a few migration frameworks (Paper A) for reactive and extractive adoption, however, those frameworks significantly rely on heuristics and developer intervention. As a result, they are limited in accuracy and completeness, rendering them less effective and error-prone in practice. Contrary to those frameworks, our framework relies on explicitly recorded metadata for enabling automated change propagation and SPL-migration. We prescribe proactive logging of features and feature mappings in assets. Virtual platform is language-independent, and can be incorporated in multiple ways. The evaluation of our framework features a simulation study of the development of a real-world project, and the results imply that using virtual platform saves the costs of clone detection and feature location. We also study the potential of features beyond code-level; investigating the impact of using different feature representations (variability mechanisms) on the the comprehensibility of common model-related tasks. Our results show that variability mechanisms have different impacts on model comprehensibility depending on the nature of the task and the type of model.

As part of our ongoing work, we aim to conduct a user-study to investigate the accuracy and usefulness of using the virtual platform in real-time. To this end, we are building plugin-support for our framework on top of HAnS (Helping Annotate Software) [42], an IntelliJ IDE plugin that supports developers to efficiently record features as they code. Additionally, in an effort to standardize the notation for adding features in a language-independent manner, we combine the existing approaches for specifying embedded feature annotations and provide a unified notation, as well as a plugin for extracting annotations specified in our notation (Paper a). Lastly, we aim to incorporate *safe evolution* [43] operators for product-line evolution in the virtual platform. While virtual platform provides basic support for verifying the validity of the structures (`Well-formedness` criteria), additional support is needed for more complex operations (e.g., splitting an asset or making a feature optional).

Chapter 2

Paper A

Seamless Variability Management With the Virtual Platform

Wardah Mahmood, Daniel Strüber, Thorsten Berger, Ralf Lämmel, Mukelabai Mukelabai

In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) (pp. 1658-1670). IEEE.

Abstract

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

2.1 Introduction

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

Two opposing strategies exist for engineering variants. A convenient and frequent strategy is *clone&own* [3, 44–47], 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 [3, 46, 48] 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) [2, 15, 49–51]. Individual variants are then derived by configuring the platform. This strategy is typically advocated for systems with many variants, such as software product lines (e.g., automotive/avionics control systems and industrial automation systems) or highly configurable systems (e.g., the Linux kernel). This strategy scales, but is often difficult to adopt and requires substantial up-front investments, since variability concepts (e.g., a feature model [52, 53], feature-to-asset traceability [54, 55], a configuration tool [56]) 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 [4, 48, 57, 58], recovering meta-data that was never recorded during clone&own, such as features and their locations in software assets [53, 59].

Over the last decades, researchers focused on heuristic techniques to recover information from legacy codebases, including feature identification [60–62], feature location [63–65], variability mining [66, 67], and clone-detection techniques [68, 69]. Unfortunately, such techniques are usually not accurate enough to be applicable in practice, and also require substantial effort to set them up and provide with manual input (e.g., specific program entry points for feature location techniques [37]). As we will show, existing platform migration techniques either heavily rely on such heuristics or have only been formulated as abstract frameworks so far. Moreover, they tend to prescribe non-iterative, waterfall-like migrations, making it risky and expensive.

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

decide to use only a subset of all the variability-implementation concepts that are typically required for an integrated platform. This allows organizations to be flexible and innovative by starting with clone&own and then incrementally adopting the variability-implementation concepts necessary to scale the development, as indicated by industrial practices for product-line adoption [51, 70–72]. 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 [1] 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 [57] often leading to re-engineering efforts over years [4, 73]. This early, high-level description of a strategy to incrementally scale the management of variants paved the way for this paper. One of our core contributions are conceptual structures and formalized operators for the virtual platform, which are related to ordinary code editing, but also record and exploit meta-data. While we prototypically implemented the virtual platform on top of an ordinary file system, our work gives rise to realize it upon a database (to enhance scalability), within an integrated development environment (IDE), or as a command-line tool. The meta-data could also easily be saved directly in the software-assets using lightweight embedded annotations (as our prototype does). We evaluated our prototype on a reasonably sized system (57.4k lines of text, 4 variants), where we simulated evolution activities that are typical of practical software systems. Our prototype was able to fully simulate and manage all considered activities. From a cost-benefit analysis, we conclude that the virtual platform offers significant cost savings during inevitable evolution and maintenance activities.

In summary, we contribute:

- **a mechanization** of the so-far abstract idea of operators mediating between clone&own and an integrated platform, defined upon conceptual, language-independent structures,
- **a prototype of the virtual platform [74]** in Scala,
- **a comparative evaluation of the virtual platform** against five related frameworks, based on their ability to support common evolution scenarios,
- **a cost-and-benefit evaluation of the virtual platform**, based on a simulation study featuring the revision history of a real variant-rich open-source system, and

- **an online appendix [75]** with a technical report about our operators, additional examples, and evaluation data.

2.2 Motivation and Overview

We provide a core scenario of seamless variability management as a running example and an overview of the virtual platform. While rooted in a deliberately simple application domain, the example is inspired by documented real product-line migration scenarios [58, 76]. It includes tasks that are tedious and error-prone in practice (e.g., bugfix propagation along branches).

2.2.1 Motivating Running Example

We now discuss relevant problems of managing variants inspired by actual industrial practices, also presenting our solution in the virtual platform and how a developer would use the virtual platform. Specifically, developers interact with the virtual platform by invoking its provided operators, either via the command-line or an integration with an IDE or version-control system provided by a tool vendor (see Section 2.2.2 for details). While the traditional, asset-oriented operators can run transparently in the background, only the feature-oriented operators require an extra user interaction for invoking the operators. The operator are described in detail in Section 2.5.

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

Figure 2.1 illustrates the two opposing strategies (cf. Section 2.1) 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 [27, 77]), as well as the representation of variability in the integrated platform. We also need more variability concepts, among others, features [78–80], code-level configuration [15], feature-to-asset traceability [54, 55, 81], a feature model (a hierarchical structure with features and their dependencies) [52, 53], a configurable build system [15], and a configurator tool [56, 82, 83]. This example shows that, when it becomes necessary to migrate from clone&own to an integrated platform, important information needs to be recovered, specifically: that a feature *SmallDisplay* was implemented and where its code is located. Recovering such information in systems with many features and sizable codebases is laborious, time-consuming, and inaccurate at best. Also, migration can be invasive, risky, and costly, especially hard to achieve under market pressure [4, 57, 59, 73, 84, 85].

The virtual platform exploits a spectrum between the two extremes and supports an incremental transition as shown in Figure 2.2. It adapts the

governance levels from prior work [1], 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. (Figure 2.2, 1st level). Our solution is to record traceability information about the cloned variants' provenance, which eases tracking and synchronizing clones. It also bypasses the inaccuracies associated with clone detection, making tasks such as change propagation more effective. The virtual platform records clone traces among assets in the background, without requiring extra effort from the developer, but who can query it for obtaining the clones of an asset.

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

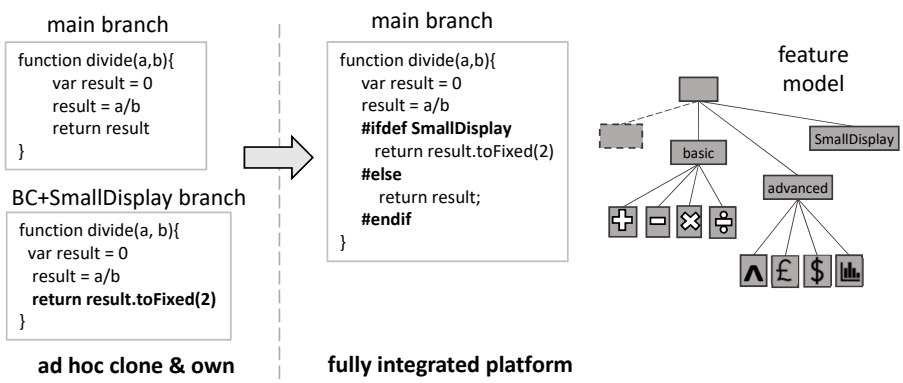


Figure 2.1: Ad hoc clone&own vs. fully integrated platform illustrated for two variants: the *BasicCalculator* and a variant with only a small display

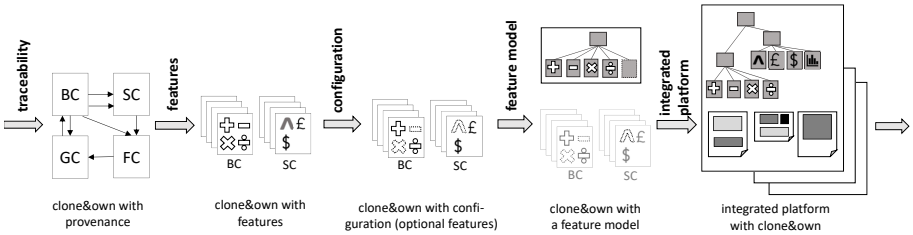


Figure 2.2: Spectrum between the extremes ad hoc clone&own and a fully integrated platform (see Figure 2.1 for both), illustrated with cloned variants: *BasicCalculator* (BC), *ScientificCalculator* (SC), *GraphingCalculator* (GC), and *FinancialCalculator* (FC). The virtual platform provides operators to transition along this spectrum (e.g., to incrementally adopt a platform).

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. (Figure 2.2, 2nd level). Adding feature meta-data adds perspective and allows functional decomposition. It also allows representing assets in terms of features, to reuse and clone features across projects. Lastly, including feature-related information allows going past the efforts and inaccuracies of *feature location* (recovering where a feature is implemented), making feature reuse and maintenance more effective. The virtual platform offers operators to add features conveniently (at the same time annotating assets).

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

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

Solution 3: Clone&own with configuration. (Figure 2.2, 3rd level). To reduce it, our organization starts to incorporate configuration mechanisms. These allow to enable or disable features, such as *SmallDisplay*, which control variation points. This reduces redundancy and maximizes reuse. So, the organization maintains a list of features and uses a configurator tool. The virtual platform supports this solution with a simple operator.

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

Problem 4: How to keep an overview over the features? The more features and variation points the organization incorporates, the more it loses overview over the features and their relationships, including feature dependen-

cies (accidentally ignoring those can lead to invalid variants). Maintaining such information would also help scoping variants.

Solution 4: Clone&own with a feature model. (Figure 2.2, 4th level). Our organization introduces a feature model, which captures features and their constraints, also as input to the configurator. Feature models are very intuitive and simple models, which provide deep insights without much additional tool support. They also foster communication among stakeholders and validate feature configurations. With this solution, consistency between features and clones is high, since developers can also exploit the clone traces and use the virtual platform for feature-based change propagation.

The developer adds a feature model to the repository with the operator `AddFeatureModelToAsset`. She can change the feature model to add and remove features at any 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. (Figure 2.2, 5th level). Our organization integrates the projects into a consolidated platform. Luckily it can exploit meta-data about clone traceability (*provenance*) and features with their locations in assets. The virtual platform provides support for this kind of information, easing the integration of cloned variants into a platform. Of course, developers might have forgotten to record all that information, then it is natural to recover it. As long as some information is recorded, a benefit arises in terms of saved feature identification, feature location and clone-detection effort.

2.2.2 Virtual Platform Overview

Our goal is to combine the benefits of the two opposing strategies clone&own and integrated platform, exploiting a spectrum between both and allowing incremental transition as in our running example (Section 2.2.1). To this end, we designed a framework called virtual platform comprising conceptual structures upon which operators modifying the structures are executed by developers. The conceptual structures abstractly represent software assets at various levels of granularity—from whole repositories to blocks of code—and can be adapted to specific asset languages (explained shortly in Section 2.4). In addition, they maintain information about variability, specifically feature information, feature-to-asset mappings, and clone traces. The virtual platform extends other development tools, specifically, IDEs and version control systems. On top of these, which are concerned with the management of *assets*, the virtual platform provides dedicated functionality for managing *features*. Operators can be either traditional, meaning they are concerned with asset management, or feature-oriented, meaning they are devoted to features and their locations in assets. In contrast to traditional development workflows, the

use of dedicated feature-oriented operators incurs a certain cost, but promises benefits to developers. In Section 2.6, we study this trade-off.

Figure 2.3 illustrates interactions and internal workings of the virtual platform. Developers can interact with it directly or indirectly. The former is enabled via extensions and hooks of existing tools. Specifically, traditional IDE commands such as *"Create File"* and version-control commands such as *"Add File"* are linked to the traditional, asset-oriented operators of the virtual platform (e.g., *"Create Asset"*) and do not impose additional effort for developers. Feature-oriented operators can be implemented by new, feature-oriented IDE commands (e.g., *"Create Feature"*). Direct interaction is enabled via a command-line interface, where developers can call feature-oriented operations such as *"Create Feature"* directly.

2.3 Methodology

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

Initial Design.. We started by analyzing clone-management and platform-migration frameworks proposed in the literature, from which we extracted development activities that should be supported by the virtual platform. We also had a series of discussions among the authors, one from industry and four from academia. Two authors have over ten years of research experience in variability management and SPLE. We also created ad hoc examples in the discussion meetings. From these sources, we identified an initial set of data structures and operators, and implemented them in Scala.

Specifically, from the literature, we identified five relevant works on clone management and product-line migration using our expert knowledge. Rubin et al.'s product-line migration framework [5, 10] offers operators that support the narrative that a mechanization—i.e., an operator-based perspective—leads to more efficient implementation and support. Fischer et al.'s [6] 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 [86] is an extraction-based technique for product-line migration, including support for feature-model synthesis. Pfofe et al.'s tool VariantSync [8] supports clone-management by easing the synchronization of assets among

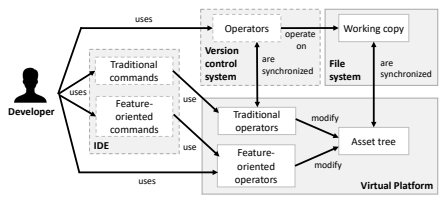


Figure 2.3: Overview (dashed boxes represent optional parts)

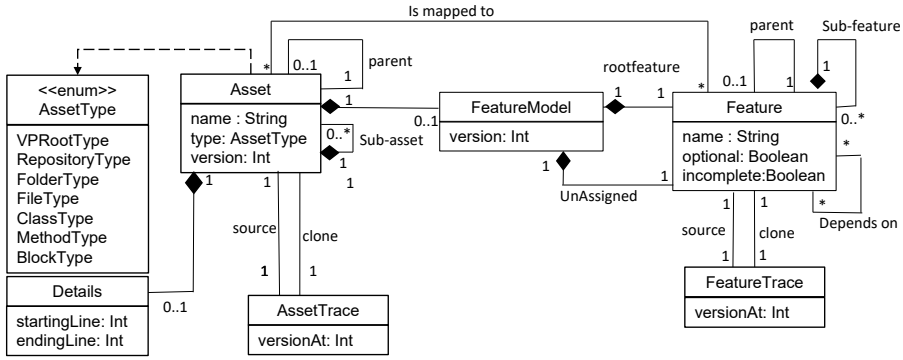


Figure 2.4: Conceptual structures: asset tree, features, mappings, and clone traces

cloned variants. Montalvillo et al.'s operators and branching models for clone management in version-control systems [9] allow isolated variant development with change propagation, but without using the notion of features, as opposed to the other frameworks. For brevity, we will present the identified activities only at the end in Section 2.6.1. Detailed descriptions are in our online appendix [75].

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

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

2.4 Conceptual Structures

The virtual platform's conceptual structures form the basis for its operators, which we formulated as functions with side effects (in-place transformations) that modify the structures. Figure 2.4 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. Section 2.6).

Asset Tree (AT). is our main conceptual structure and abstractly represents a hierarchy of assets, such as the folder hierarchy, but also the hierarchy within source files. In Figure 2.4, 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, [87]), which represent source 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 files. Every node represents an `Asset` related to the project, such as a folder, a file (e.g., image, source file, model or requirements document), or text. Every `Asset` has a `name`, a type (`AssetType`), and a `version` (a simple means to identify changes). An asset can have any number of `sub-assets`. It also owns a `parent` pointer p, which should define a tree, with a virtual root node (`Asset` of type `VPRootType`) denoted as `root`. The `AssetType` is used to capture the role of the `Asset` in the project, and can be one of the following: `VPRootType`, `RepositoryType`, `FolderType`, `FileType`, `ClassType`, `MethodType`, and `BlockType`. The type `VPRootType` is only used once in the AT, to specify the synthetic root node. The main purpose of this root node is to carry a global version (we explain versioning shortly).

Traditional SPLE architectures have a feature model per project, which can be difficult to maintain and evolve in large systems (e.g., Linux kernel [88]). 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 Figure 2.4).

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

Features and Feature Models. A `feature` has a `name` and two `Boolean` parameters: `optional` and `incomplete`. The field `optional` specifies whether the `feature` is mandatory or optional; `incomplete` captures information about the completeness of the *feature's* implementation. If the `feature` was cloned from another `feature model` scope, it is `true` if the new scope containing the `feature` also contains all the assets to which the `feature` is mapped; otherwise it is always `false`. Every `feature` has an optional `parent`, and any number of `sub-features`. Features can have dependencies to each other. A `feature model` has a *root* `feature` and a mandatory `feature` called *Unassigned*, which contains all features that are added to the model as a result of asset cloning. That is, if any `feature` mapped to the `Asset` is not present in the target `feature model` already, it is mounted under *Unassigned*(and requires developer intervention to move it to the desired location in the model).

Asset-To-Feature Mappings., in practice, can have two semantics. They can be simple mapping relationships, indicating that `Asset` realizes a `feature` [16]. They can also indicate variability [89], where the `Asset` is included in a concrete variant if the `feature` is selected (interestingly, if an `Asset` is optional based on a `feature`, then the `Asset` also realizes it, but not necessarily all assets realizing a `feature` are optional). The SPLE community usually focused on the variability relationship, and the feature-location community on traceability. For the virtual platform, we unified the mechanism with which assets are mapped to features. Specifically, an `Asset` has a presence condition (PC)—a propositional formula over features. A PC allows conveniently mapping assets of different granularity levels (`AssetType`) to entire `feature` expressions. Whether this relationship to the `feature` represents variability or traceability is solely determined by the *feature's* `optional` parameter.

Versioning of Assets.. Assets (and features) have a `version`—an integer

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

Clone Traceability.. To maintain trace links between source assets and their clones, we define an `AssetTraceDatabase`—essentially a list of `AssetTraces` (Figure 2.4). An *AssetTrace* is a triplet of the source `Asset`, its clone, and a `version` at which the source `Asset` was cloned. Similarly, `feature` traces are used to keep track of the `feature` clones, and they are stored in a `FeatureTraceDatabase`. A *FeatureTrace* is also a triplet pointing to the source `feature`, its clone, and `version` at the time of cloning. These traces are a core component of our contribution, and have special relevance in cloning and change propagation for both assets and features. For brevity, we refer to both `AssetTraceDatabase` and `FeatureTraceDatabase` as *TraceDatabase* in the remainder of the paper.

2.5 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 [75]. The appendix also presents a number of additional *convenience operators*—utility methods that efficiently traverse the trees (AT and `feature model`) to return data that needs to be frequently accessed (such as assets mapped to a `feature` and clones of an `Asset` etc).

2.5.1 Traditional/Asset-Oriented Operators

We represent conventional activities performed by developers using asset-oriented operators. These operators allow to keep the AT in sync with the working directory. Also, the assets act as *mappable* components to the features, and allow cloning and change propagation. In what follows, we introduce the asset-oriented operators with their parameter types, a brief description, and sample scenarios, inspired from our calculator running example (cf. Section 2.2.1). The notation used for visualizing various scenarios is shown in Figure 2.5.

`AddAsset` : `Asset` $\times$ `Asset` $\rightarrow \mathbb{B}$

Description: When a source `Asset` (S) is added in any target `Asset` (T) to a repository (e.g., a file to a folder), `AddAsset` creates an `Asset` for S and adds it to the preexisting `Asset` T in the AT. Additionally, it increments the

`globalVersion`, and assigns it to S and T. This implies that the most recently changed assets are S and T. Also, it adds any `feature` mapped to S in T's `feature model` (typically repository `feature model`).

Example: Consider the *BasicCalculator (BC)* example. The developer adds the implementation for the *divide* method in the file *Operators.js*, with an annotation for the `feature` *DIV*. Consequently, the virtual platform creates and adds the `Asset` *divide* (S) of `MethodType` to the `Asset` *Operators.js* (T) of `FileType`, and *DIV* to the `feature model` of T. The `globalVersion` (previously 3) is incremented and assigned to *divide* and *Operators.js*. Figure 2.6 illustrates the scenario.

`ChangeAsset` : `Asset` $\rightarrow \mathbb{B}$

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

`RemoveAsset` : `Asset` $\rightarrow \mathbb{B}$

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

`MoveAsset` : `Asset` $\times$ `Asset` $\rightarrow \mathbb{B}$

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

Thus far, the operators we presented serve two purposes: keeping the *AT* synchronized with the project, and keeping track of changes through versioning. Following, the operators serve two additional purposes: storing feature-oriented data, and recording traceability among clones. The exploitation of these meta-data are the essence of our framework.

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

Description: Upon addition of a `feature` mapping by a developer, `MapAssetToFeature` checks if the `feature` exists in the `feature model` of the `Asset`. If not, it creates a `feature` F (with the name used by the developer), maps it to S (corresponding `Asset` in the *AT*), and adds F to the *Unassigned* `feature` in the `feature model` of S. If F already exists, it simply maps F to S. For

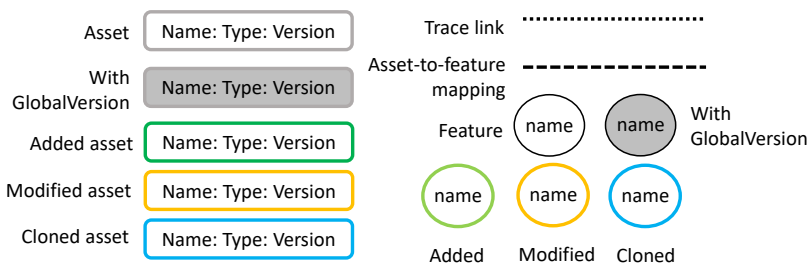


Figure 2.5: Notations used in operator illustrations

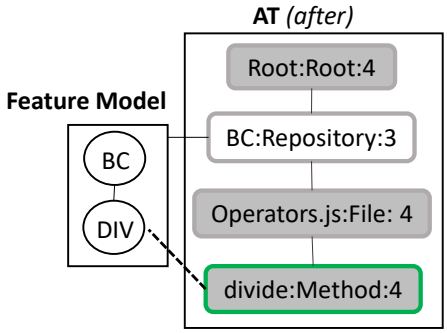


Figure 2.6: Illustration of `AddAsset`(*divide*,*Operators.js*)

mapping, it adds F to the `presencecondition` of S with a logical disjunction. To track this change, the `globalVersion` is incremented and assigned to S.

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

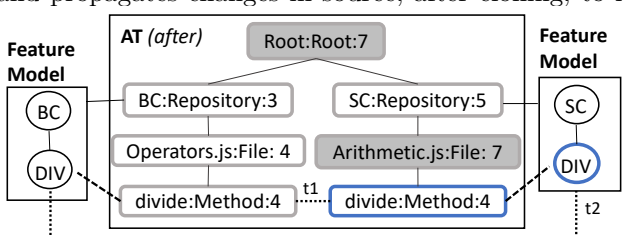
`CloneAsset` : `Asset` $\times$ `Asset` $\rightarrow \mathbb{B}$

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

Example: Starting from Figure 2.6, the developer copies the method *divide* in *Arithmetic.js*; a file in another project, *ScientificCalculator (SC)*. `CloneAsset` clones *divide* to *Arithmetic.js*, an `Asset` of `FileType` in *SC*, as well as the mapped `feature` *DIV* in the `feature model` of *SC*. Traces for both *divide* and *DIV* are added to the *TraceDatabase*. Figure 2.7 illustrates the scenario.

`PropagateToAsset` : `Asset` $\times$ `Asset` $\rightarrow \mathbb{B}$

Description: `PropagateToAsset` takes two assets, checks if one is a clone of the other, and propagates changes in source, after cloning, to its clone. To



seq	source	clone	versionAt
t1	divide	divide	4
t2	div	div	1

Trace Database

Figure 2.7: Illustration of `CloneAsset`(*divide*, *Arithmetic.js*)

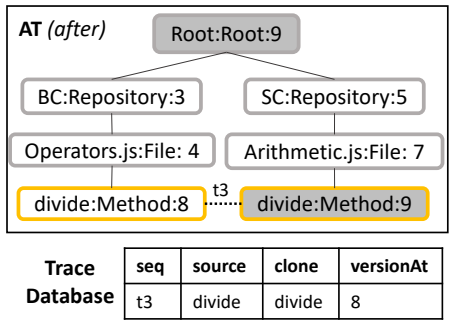


Figure 2.8: Illustration of `PropagateToAsset`(*divide*, *divide*)

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

Example: Assume that the *divide* method during cloning did not include the check for division by zero. After adding the check (`ChangeAsset`), the *divide* method in source (*Operators.js*) is ahead (*version*=8) of the *divide* method in target (*Arithmetic.js*), with *version*=4. By invoking `PropagateToAsset`, the changes are propagated automatically. Figure 2.8 demonstrates the scenario; for simplicity, `feature` mappings are omitted.

2.5.2 Feature-Oriented Operators

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

`AddFeature` : `feature` × `feature` → $\mathbb{B}$

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

Example: Assume that the `feature model` for *BC* is a textual file, where features are written as individual lines, and indentation is used to represent hierarchy (Clafer syntax [90]). The developer adds a line "EXP" (exponent), below the line "BC" (*root* feature, *BC)*. `AddFeature` creates a corresponding `feature` *EXP*, and adds it to the `feature` *BC*. The `version` of *root* `feature` is incremented (previously 1 after adding `feature` *DIV*) and assigned to `feature` *EXP*. Figure 2.9 demonstrates the scenario, with the resulting versions in a table on the right.

`AddFeatureModelToAsset:` `Asset` × `feature model` → $\mathbb{B}$

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 `AddFeature-`

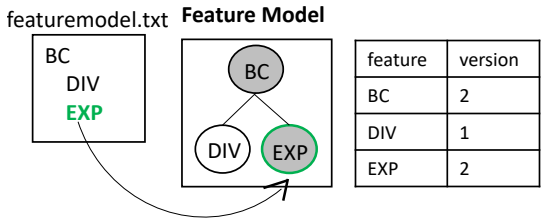


Figure 2.9: Illustration of `AddFeature`(EXP, BC)

`ModelToAsset`. The operator then locates the `Asset` in the AT, creates a `feature model` FM, and sets the asset's parameter `feature model` to FM. The `globalVersion` of the AT is incremented and assigned to the `Asset` which contains FM.

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

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

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

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

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

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

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

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

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

Example: After adding the `feature` EXP (using `AddFeature`), the developer

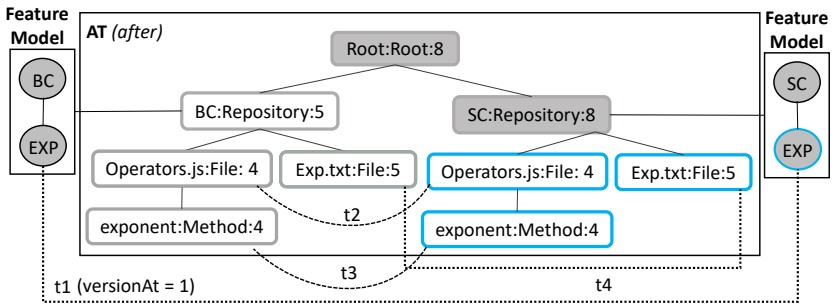


Figure 2.10: Illustration of `CloneFeature`(*EXP*, *SC*)

added two assets in `feature` *BC*, and later mapped them to `feature` *EXP*. The assets are a method "*exponent*" and a textual file "*exp.txt*" with documentation of *exponent*. The developer now wants to reuse `feature` *EXP* in *SC*. To clone the feature, she invokes `CloneFeature`, which clones the `feature` *EXP* and its mapped assets to *SC*. Additionally, traces for the `feature` and asset clones are added to the *TraceDatabase*. This example is illustrated in Figure 2.10. Note that even though *Operators.js* was not cloned, the virtual platform created a clone, as the method *exponent* could not be added directly to the repository. This is referred to as *tree slicing*, which the virtual platform adopts to ensure that the `well-formedness` of the *AT* is maintained.

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

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

2.6 Prototyping and Evaluation

We prototyped and evaluated the virtual platform qualitatively and quantitatively: (i) in a comparative assessment against the frameworks presented in Section 2.3, (ii) 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 [75]. The prototype, implemented in Scala, provides an API as the main interface to execute the operators. In the production-ready tool, this API would be usable as a command line interface or a set of IDE commands. We used a strategic programming library (kiama) for efficient tree traversal and rewriting. After implementing all operators, we created test scenarios to verify the correctness. These test scenarios were developed using domain knowledge acquired by experience, and also inspired by observing scenarios from the case study of Clafer Web Tools. We checked correctness by comparing the result state (AT, trace, and mappings) after operator invocations to the expected one. We also simulated the illustrative example presented in Section 2.2.1 by automatically realizing all the discussed scenarios.

2.6.1 Comparative Evaluation

For comparison, we extracted activities supported by techniques for supporting clone&own (a.k.a., clone management), or the migration of cloned variants to an integrated platform (a.k.a., product-line migration). In total, we extracted 12 activities which we found to be common across most, if not all, existing techniques. We evaluated the virtual platform's ability to support the scenario from Section 2.2 and the 12 activities of related frameworks. Details are in the appendix [75].

Table 2.1 shows whether and how an activity related to either clone management or product-line migration is supported by an existing framework, as well as virtual platform. The activities are: feature identification (features defined in a variant), feature location (recovering traceability between features and assets), feature dependency management (managing constraints among features), feature model creation (creating and evolving a feature model), storing feature-to-asset mappings, clone detection (identifying assets which are clones of one another), feature cloning (ability to clone features), change propagation (replicating changes made in an asset to its clone), creation of reusable assets (which can be used to derive variants), product derivation (ability to derive a partial or complete product given a configuration), variant integration (merging assets/variants by taking variability into account), and variant comparison (comparison of assets to find commonalities and varaibilities).

In summary, among all frameworks, the virtual platform is the first one

Table 2.1: Comparison of the virtual platform with activities supported by clone-management and product-line migration frameworks

Feature identification → abstract operator [10], specified in the beginning [6, 8, 86], specified any time in virtual platform
Feature location → abstract operator [10], extracted [6, 86], internal tagging [8], also internal tagging in virtual platform
Feature dependency management → abstract operator [10], statically mined [86], specified in beginning [8], specified any time in virtual platform
Feature model creation → multiple abstract operators [10], activity [86], specified in the beginning [8], dynamically grows in virtual platform
Feature-to-asset mapping → abstract operator [10], extracted [6, 86], specified any time [8], specified any time in virtual platform
Clone detection → textual diff tools [10], feature expression comparison [8], git clone points to source [9], not needed in virtual platform
Feature cloning → supported by virtual platform
Change propagation → multiple abstract operators [10], variant synchronization [8], using Git merge [9], automated in virtual platform
Reusable assets creation → abstract & incremental [10], reuse existing variants [6], reusable core assets [9, 86] and features in virtual platform
Product derivation → abstract [10], customizing after cherry-picking [9], composition [6, 86], preprocessor-like in virtual platform
Integration → abstract operator using meta-data [10], third party tool [8], Git merge [9], manual or tool-based, guided by meta-data in the virtual platform
Variant synchronization → Git dif [9], code comparison [6, 86], not needed in virtual platform

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

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

2.6.2 Simulation Study

We used an open-source system called Clafer Web Tools (CWT, [91]) that was evolved using clone&own in three cloned variants (*ClaferMooVisualizer*, *ClaferConfigurator*, *ClaferIDE*) towards an integrated platform (*ClaferUICommonPlatform*), including many feature clonings across the variants. We evaluated the virtual platform's efficiency by simulating the evolution of CWT, retrofitting our operators to achieve the original evolution, and studying the costs and benefits.

We used a dataset by Ji et al. [16] 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 [75].

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

Cost & Benefit.. As costs, we measure the additional effort imposed on developers by our platform. Our traditional, asset-oriented operators (left-hand column of Table 2.2) 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 2.2): 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

Table 2.2: 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

changes along clones from our clone database.

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

RQ1.. *What are the costs of maintaining features using feature-oriented operators?* The overall cost C_{feat} arises from accumulating the cost of applying feature-oriented operators. Each feature-oriented operator *op* has a cost $C_{feat}(op) = \#invoc(op) * cost_{abs}(op)$, which depends on the number of invocations of *op*, and the absolute cost of each invocation of *op*. Based on Table 2.2, 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 required additional feature location effort?* The omission-related cost C_{miss} arises from the number of late invocations of `MapAssetToFeature`, representing situations where the developer missed to specify an asset-to-feature mapping when the asset was added. This number is to be multiplied by the absolute cost for these invocations, which is generally higher than a regular invocation. Our operators `CloneFeature`, and `PropagateToFeature` rely on a complete mapping from a feature to its assets. A third relevant operator is `AddFeature` which adds feature information to source code added earlier. In absence of a recorded mapping, each operator requires an expensive manual feature location step, which is not required in our approach (see RQ3). We counted the number mappings that were added before or after one of these operators was invoked, which indicates that the researcher preparing the original dataset noticed an omission. We determined 14 relevant mappings for `CloneFeature` (2 relevant invocations, 3.7% 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 [16] 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 [92]. Tool-based clone detection requires manual verification and postprocessing, since even the most advanced clone detection tools have imperfect precision and recall [93].

2.6.3 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 perform an approximation based on plausible estimates: (1.) For the cost of feature location, we rely on the earlier literature estimate [16] of 15 minutes per instance. (2.) We assume clone detection to have the same cost as feature location. (3.) We assume the cost for adding an omitted annotation to be 10 times as high as a regular operator invocation. Based on these three assumptions, we break even if *invoking a feature-oriented operator takes 54 seconds or less* on average. In practice, the benefit can be assumed to be larger, since invoking a feature-oriented operator mostly entails picking a feature name (while the feature is still fresh in the developer's mind), a matter of a few seconds.

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

Representativeness.. Our case is representative for systems of comparable size (547k lines, four variants). Many reported product-line migrations are of similar size [94]. We argue for representativeness for larger systems qualitatively. Our case has all evolution activities observable in industrial systems, supported

by other frameworks. Still, the virtual platform is evaluated more extensively than any of these.

Threats to Validity.. A threat to external validity is that our operators do not completely capture the real-world scenarios developers encounter when dealing with variant-rich systems. We mitigate this threat with our evaluation based on the simulation of a real system. There is a general lack of available systems for benchmarking on realistic revision histories with available feature information, a problem that we aim to address as part of our ongoing benchmarking initiative [95, 96].

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 [97, 98]). 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.

2.7 Related Work

The five most closely related works are the clone-management and product-line-migration frameworks that we used to inform the virtual platform's design ([6, 8–10, 86], cf. Section 2.3). In Section 2.6.1 and our online appendix [75], we provide a detailed comparison, highlighting unique benefits of the virtual platform: support for early traceability recording, operators for the full spectrum between the extremes ad hoc clone&own and integrated platform, and an evaluation on a real project revision history. We now discuss further related work on product-line migration and integrated-platform evolution.

The idea of automatically handling variation points, as the virtual platform does, is not new. In fact, going back to the 1970s, researchers have built so-called variation-control systems [99, 100], which never made it into the practice of software engineering. These systems have been realized upon different back- and frontends (e.g., version-control systems [101, 102] or a text editor [103]), 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 [58]. Case studies of manual migration [4, 57, 59, 84, 104, 105] 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 [104–107], typically

proposing refactoring techniques. Wille et al. [108] use variability mining to generate transformational rules for creating delta-oriented product lines.

Others focus on evolving software platforms. Liebig et al. [109] present variability-aware sound refactorings (rename identifier, extract function, inline function) for evolving a platform by preserving the variants. Rabiser et al. [110] present an approach for managing clones at product, component, and feature, and define 5 consistency levels to monitor co-evolving clones. Ignaim et al. [111] present an extractive approach to engineer cloned variants into systematic reuse. Neves et al. [43] 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.

2.8 Conclusion

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

We see several promising directions of future work. By allowing developers to continuously record feature meta-data, the virtual platform paves the way for software analyses that rely on this data. One example is support for the safe evolution of product line platforms [43], which could be extended to support systems in our intermediate governance levels. Specifying our operators in the framework of software product line transformations [112–114] would make them amenable to conflict and dependency analysis [115], 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 [32], we could facilitate inspection of the available feature mappings. Offering a "preview mode" would allow to inspect and interact with the changes arising from a planned operator invocation. Providing a dedicated operator to integrate cloned features is another future direction. Other directions are to support configuration of

variants by selecting features, offering views [116], and providing visualizations (e.g., dashboards [117,118]). Finally, recommender systems that learn from the meta-data and support developers handling features and assets could further encourage using features in software engineering [119].

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

Chapter 3

Paper B

Promote-pl: A Round-Trip Engineering Process Model for Adopting and Evolving Product Line

Jacob Krüger, Wardah Mahmood, and Thorsten Berger

In Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A-Volume A (pp. 1-12).

Abstract

Process models for software product-line engineering focus on proactive adoption scenarios—that is, building product-line platforms from scratch. They comprise the two phases domain engineering (building a product-line platform) and application engineering (building individual variants), each of which defines various development activities. Established more than two decades ago, these process models are still the de-facto standard for steering the engineering of platforms and variants. However, observations from industrial and open-source practice indicate that the separation between domain and application engineering, with their respective activities, does not fully reflect reality. For instance, organizations rarely build platforms from scratch, but start with developing individual variants that are re-engineered into a platform when the need arises. Organizations also appear to evolve platforms by evolving individual variants, and they use contemporary development activities aligned with technical advances. Recognizing this discrepancy, we present an updated process model for engineering software product lines. We employ a method for constructing process theories, building on the recent literature as well as our experiences with industrial partners to identify development activities and the orders in which these are performed. Based on these activities, we synthesize and discuss the new process model, called *promote-pl*. Furthermore, we explain its relation to modern software-engineering practices, such as continuous integration, model-driven engineering or simulation testing. We hope that our work offers contemporary guidance for product-line engineers developing and evolving platforms, and inspires researchers to build novel methods and tools aligned with current practice.

3.1 Introduction

Software product-line engineering provides methods and tools for building variant-rich systems. It allows to systematically reuse software features (i.e., user-visible functionalities of a system) by establishing an integrated software platform [12, 15, 120]. To build a platform, developers employ a range of implementation techniques called variability mechanisms [15, 121] to define variation points. Individual variants can then be derived through configuring—enabling or disabling features. Typical variability mechanisms comprise preprocessors (e.g., the C preprocessor), configurable build systems, configurator and variant-derivation tools [122–124], as well as model-based representations of features and their constraints, called variability models [48, 53, 125, 126]. Especially the latter are core to manage features and to guide the derivation of individual variants.

While the underlying ideas and mechanisms employed remain similar, their implementation and usage have evolved considerably over the last decades, enabling organizations to rely on more advanced automation. Examples of these advancements are novel analysis techniques for feature models, code, and test assets [95, 127–129], or the adoption of continuous integration [130, 131]. Many of these techniques have implications on the processes with which variant-rich systems are engineered. Unfortunately, the process models for product-line engineering have not been updated accordingly. Consider one of the most common process models for product-line engineering [12], as shown in Figure 3.2. This model strictly distinguishes between domain engineering (i.e., developing the platform) and application engineering (i.e., developing variants), defining five and four activities, respectively. This process model should be updated to reflect, for example, the less strict separation of domain and application engineering in practice, the evolution of product lines via their variants, and different adoption strategies [132] (we detail these examples in Section 3.2). In short, we believe that the core limitations of existing process models are the strict separation of domain and application engineering, and the focus on the proactive adoption strategy, which, as we will show, do not reflect industrial and open-source practice anymore.

We present *promote-pl* (PROcess MOdel for round-Trip Engineering of

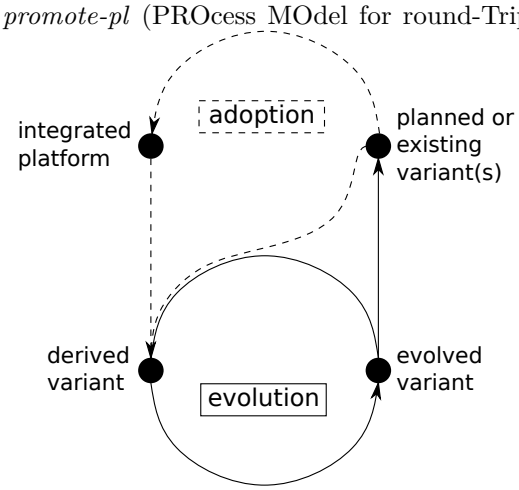


Figure 3.1: High-level representation of promote-pl.

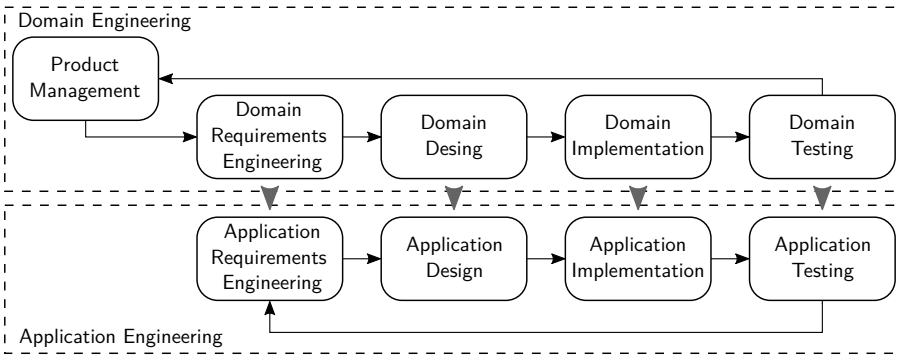


Figure 3.2: A typical product-line process model [12].

Product Lines), an updated process model for product-line engineering that we synthesized from recent literature and collaborations with industry. For this purpose, we adapted methods for deriving process theories [133,134] to elicit empirical data and synthesize promote-pl (high-level representation in Figure 3.1, explained in Section 3.4.2). We further discuss the adaptations we implemented in promote-pl and analyze its relations to contemporary software-engineering practices to define research opportunities. In detail, we contribute:

- A systematically elicited process model for product-line engineering that reflects recent practices, called *promote-pl*.
- A discussion of the adaptations we implemented and their implications for practice as well as research.
- An analysis of relations between promote-pl and software-engineering practices.

With these contributions, we intend to provide a more realistic and updated process model for product-line engineering that can provide a better understanding of how organizations engineer software platforms. Especially for researchers, promote-pl highlights the differences between historically defined process models and industrial practices, helping them to identify research opportunities.

3.2 Motivation and Objectives

In Figure 3.2, we illustrate the structure of a typical product-line process model [12]. We can see that the domain engineering encompasses a special activity for product management as well as activities for requirements engineering, design, implementation, and testing of the platform. Moreover, there is a loop between these activities, indicating evolution of the platform. Strictly separated and always building on the defined platform is the application engineering with the respective four activities for deriving a variant. This process model is a high-level abstraction (i.e., [12] describe the activities in more detail, as well as the need to tailor the model to concrete systems) that is similar to other established process models, most notably those of [11], [13], [14], and [15]. However, even though some of these process models encompass minor differences, they appear to not be in line with contemporary practices [51], they largely disregard recent trends of blurred boundaries between software-engineering phases

(e.g., continuous software engineering [135]), and they focus on the proactive adoption strategy (explained shortly). For example, [12] suggest to conduct the "Commonality analysis first," supporting the proactive adoption—that is, first establishing a platform before individual variants are derived. In contrast, our and others' experiences with practitioners [48,58,136] show a dominance of reactive and extractive adoption strategies, where organizations start with one or multiple variants first, before eventually establishing a platform. This means that, for instance, activities such as variability analysis are performed later [53] than prescribed by the traditional process models. Furthermore, platforms are typically evolved via variants—customers request additional features, which are first introduced into variants and later integrated (e.g., back-propagated) into the platform [46]. In this paper, we describe promote-pl as an updated process model for product-line engineering to reflect contemporary practices as well as adoption and evolution processes that are predominant in current practice [46,58,59,137].

3.2.1 Example Limitations of Process Models

We exemplify (mingled) limitations of existing process models, quoting insights from the company Danfoss with its long-living and well-documented [4, 73, 138,139] product line of frequency converters in the power electronics domain. The experiences are largely in line with our own experiences from studying industrial practice [46,51,53,59,78,80,136,140–145].

Separation of Domain and Application Engineering. We experienced that most organizations and developers do not strictly separate (or even distinguish) domain and application engineering. Instead, there is constant interaction between both. For example, features are often implemented in a variant and later integrated into the platform (see second example), for reactive and extractive adoption the platform is even defined based on existing variants (see third example), and processes iterate between platform and variant (e.g., during testing). Similarly, Danfoss experienced [73]:

> "[...] there was no strict separation between domain and application engineering in the product projects [...]"

In their case, the main idea was to limit the number of changes required to adopt processes and tool chains; facilitating an extractive adoption. So, we argue that we need a new process model that integrates interactions between domain and application engineering.

Evolution of the Product Line. Existing process models define that new requirements are propagated to the domain engineering, features are implemented on the platform, and the variant is derived afterwards. This is the ideal scenario, but most organizations and open-source projects use the well-known concept of feature forks [44, 46, 80] to implement new variants or platform features. By re-integrating these forks, the platform is evolved—but this is driven by developing and merging complete or partial variants. Obviously, missing to re-integrate the variants results in clone&own development instead of product-line engineering. Still, Danfoss experienced that feature forks allow

that [73]:

> "[...] projects could keep their independence by introducing product-specific artifacts as new features. Later on, when a change was assessed, there would be a decision on whether the change should be applied to other products and thus should be integrated into the core assets."

The stated independence and also fast delivery are benefits of feature forks, and they align with continuous integration. Due to their practical importance, different evolution scenarios should be added to the process model.

Adoption Strategies. Even though, extractive and reactive adoption strategies are more common in practice [48], existing process models focus on the proactive adoption in which a product line is planned from scratch. However, due to the economic investments and risks [46, 146–149], most organizations start with clone & own and only later migrate towards a product line. Consequently, they have a variety of existing variants from which they can, for example, recover architectures, reuse code, or analyze domain documentation to design the platform. This results in adopted, new, and re-ordered activities, depending on the adoption strategy. For instance, Danfoss [73] employed an extractive adoption, during which the organization migrated 80 % of the code and introduced continuous integration within the first five years, but:

> "Introducing pure::variants and establishing feature models for both code and parameters, and finally including the requirements, would take another two years."

As we can see in Figure 3.2, this conflicts existing process models in which a feature model is defined before the implementation (i.e., in the domain design). So, a general process model for product-line engineering should also incorporate different adoption strategies.

3.2.2 Research Objectives

Our overarching goal was to derive an updated, practice-oriented process model for product-line engineering. This model should help practitioners as well as researchers in understanding current practices, fostering the adoption, improvement, and future research of product lines. To achieve this, we defined three research objectives:

RO_1 Elicit empirical data about contemporary product-line engineering processes and activities employed in practice.

RO_2 Synthesize a common process model that puts the identified activities into a reasonable order.

RO_3 Discuss the adaptations in the process model and the impact of contemporary software-engineering practices.

According to these objectives, we defined an empirical methodology to elicit data (RO_1) and to construct the process model, promote-pl (RO_2). Promote-pl itself (RO_2) and our discussion of adaptations and practices (RO_3) represent the resulting contributions.

3.3 Methodology

Using a process model, we can describe *how* something happens in an actual, real-world process [150]. In contrast, development methodologies describe an assumed "best practice" of doing something, while a process theory is a universal description of a process [133]. We remark that researchers heavily debate about what a process theory constitutes in detail, and some definitions are close or even identical to a process model [133, 134]. However, following the distinction of [133], we define a process model, since we focus on constructing a process from empirical evidence, neither claiming that it represents best practices nor that it can explain all existing processes for product-line engineering in their entirety. As we can only cover the product-line engineering activities that we could identify, these two properties can, arguably, not be fulfilled; considering, for example, the numerous tools, implementation techniques, or testing strategies that exist. Moreover, future advances in research may require changes in promote-pl.

We are not aware of a specific guideline for constructing process models. Instead, we adapted recommendations for deriving process theories [133, 134]. As a result, we relied on three information sources:

- First, each author suggested publications based on their knowledge of the literature, without relying on a systematic search (cf. Section 3.3.1). This design resembles integrative reviews [151], which are helpful to critically reflect, synthesize, and re-conceptualize theoretical models for mature research areas—which was our research goal.

- Second, we extended the suggested publications based on a systematic literature review [38], searching manually in the last five instances of relevant venues (cf. Section 3.3.2). Our goal was to more systematically and extensively cover the most recent developments in product-line engineering to understand, incorporate, and discuss current practices.

- Finally, we relied on our own experiences (also adding the corresponding publications) of collaborating with industrial partners that employ product-line engineering (cf. Section 3.3.3). We used our experiences to structure our data, order activities, and discuss how practices are aligned with promote-pl.

By using these information sources, we base promote-pl in empirical evidence to strengthen its validity. In the following, we describe each information source in more detail, our strategy to elicit data from the publications identified (cf. Section 3.3.4), and how we synthesized the data to construct promote-pl (cf. Section 3.3.5).

3.3.1 Knowledge-Based Literature Selection

We used our knowledge of the literature and particularly from recently conducted (semi-)systematic literature reviews [46, 51, 53] to select publications. For this purpose, each author suggested publications that they considered relevant, based on a publication's topicality and relevancy for our research goal.

We discussed each suggestion based on the following **inclusion criteria**, and only incorporated a publication if we achieved mutual agreement:

IC_1 The publication is written in English.

IC_2 The publication describes activities of product-line engineering, suggesting at least one partial order (i.e., a minimum of two activities in a sequence of execution).

IC_3 The publication reports activities based on recent (i.e., five years) experiences (e.g., case studies, interviews) or synthesizes them from such experiences (e.g., literature reviews).

We performed an initial selection to scope our research, but also added publications later in our analysis.

Results. In the beginning, we selected five publications that describe well-known process models for product-line engineering (cf. Section 3.2) as baseline for our work—marked as BL in Table 3.1. We included these publications to have a foundation that we could extend and refine to construct promote-pl. Note that we included the publication of [11], due to the reported process model being well established, even though it does not fulfill IC_2 (no partial orders). Furthermore, we agreed to add 12 additional suggestions (marked with ER in Table 3.1) that cover up-to-date experiences (i.e., IC_3)–including publications with our experiences (marked with *).

3.3.2 Systematic Literature Selection

To define a more systematic foundation for the process model, we decided to perform the search and selection phase of a systematic literature review [38]. So, we did not only rely on our own knowledge, but extended our information sources using a replicable process.

Search. We conducted a manual search among five conferences (SPLC, VaMoS, ICSE, ESEC/FSE, ASE) and seven journals (TSE, EMSE, TOSEM, JSS, IST, IEEE Software, SPE); aiming to avoid the problems of automated searches [152–154]. For the conferences, we covered their last five editions of research and industry tracks, including the 2015 to 2019 (and additionally 2020 for VaMoS) editions for each. For the journals, we considered the years from 2016 to 2020, including online-first publications. We selected these time spans to consider current product-line practices for promote-pl.

To conduct the search, we used DBLP as of April 7th 2020—except for online-first publications, for which we relied on the journals' websites as of that same date. We selected major software engineering venues that employ peer-reviews and publish product-line research, ensuring the quality of included publications. While we certainly miss some publications that describe product-line engineering processes, we argue that this selection provides a reasonable overview of recent publications to understand what adaptations are required to design a contemporary process model [151].

Inclusion Criteria. To select relevant publications, we employed the same inclusion criteria as for the knowledge-based selection. Further, we essentially

added two more inclusion criteria:

IC_4 The publication has been published at the research or industry track of a peer-reviewed venue.

IC_5 The publication does not only propose a process (e.g., new testing methods), but this process is actually used in practice.

Using these criteria, we ensured that the selected publications actually cover real-world processes and not only proposals, for example, for incorporating a new research tool.

Results. With the manual search, we identified 16 new publications, which we mark with SR in Table 3.1. Note that we do not account for publications we already identified in the previous search in this set. In the end, we selected 33 publications for constructing promote-pl.

3.3.3 Industrial Collaborations

We regularly collaborate with different industrial partners that employ product-line engineering. For instance, we worked with 12 medium- to large-sized organizations to assess their state of adopting variability management [51], interviewed experts to understand feature-modeling practices [53], and collaborated with large organizations, such as Axis [46], Saab [144], or ABB, to improve our understanding of product-line practices. We used our gained knowledge, resulting publications, and ongoing discussions, to reason about the data we elicited from the literature. Particularly, we resolved unclear partial orders to construct promote-pl (Section 3.3.5) and based the discussion of software-engineering practices on this knowledge.

3.3.4 Data Extraction

For every publication, we extracted standard bibliographic data, namely authors, title, as well as publication venue and year. To construct promote-pl, we further extracted all product-line engineering activities (i.e., we did not consider "standard" activities, such as requirements elicitation) that have been mentioned in their specific wording. If these activities were in a partial order, we also extracted that order. Moreover, we extracted the scope in which these activities have been applied, for example, extractive adoption or platform-based evolution. Finally, if we identified a specific software-engineering practice to be used, we also documented this. We used a table to document and manage this data—with Table 3.1 providing a summary of that table.

3.3.5 Process Construction

To construct promote-pl, we executed the following steps:

[a] We collaboratively analyzed the process models presented in the five baseline publications (marked with BL in Table 3.1). So, we obtained an initial understanding of the existing process models, how to unify terminologies, and a first set of partial orders. However, the most important outcome was a mutual agreement on how to elicit and document partial orders.

[b] Every author suggested relevant publications, and the first author conducted the manual search.

[c] The first author read each publication, decided whether it fulfilled the inclusion criteria, and extracted the data described in Section 3.3.4 if this was the case. To ensure that we did not miss important publications or activities, the other authors verified distinct subsets of all publications.

[d] We created a list of unique activity names (150+), which the first author used to resolve synonyms, specify terms, and abstract common activities. For example, we changed all occurrences of "product" or "system" to "variant," and specified "analyze requirements" according to its context (i.e., platform, asset, or variant). The employed changes were verified and agreed upon by the other authors. We remark that we were careful and aimed not to overly abstract activities (e.g., we kept "build" as a detailed activity of "derive variant"), which is why we report 99 distinct activities in Table 3.1.

[e] We compared the different partial orders and activities based on their scope and similarities. As a result, we defined partitions of the process model (e.g., adoption and evolution).

[f] We constructed the process model by merging partial orders. To this end, the first author used re-appearing activities and similarities in the orders, structuring these according to the identified partitions. Then, we removed redundancies as far as possible to derive a unified process model.

[g] To verify and agree on promote-pl, the third author interviewed the first author. During this interview, the first author explained promote-pl, design decisions, potential alternative representations, and based on what data each model element was incorporated. We agreed to employ smaller changes in promote-pl to improve its comprehensibility and resolve unclear orders of activities.

By using this methodology, we aimed to improve the validity of promote-pl, allowing other researchers to verify and replicate it.

3.4 The Process Model Promote-pl

We describe the partial orders of activities we identified from the literature, followed by the structure and details of promote-pl.

3.4.1 Contemporary PLE Practices (RO_1)

In Table 3.1, we provide an overview of all 33 publications we considered. We can see that the publications we identified based on suggestions and the manual search cover mostly the extractive adoption strategy and evolution, which have become major topics in product-line engineering research [58, 95, 96, 172–174]. Moreover, the publications have been published in various venues, not surprisingly mostly at the flagship conference for software product-line engineering SPLC. We argue that this selection provides a broad and

Table 3.1: Overview of the 33 publications we analyzed and the activities described (based on our unified terminology).

	Ref.	Venue	Scope	Activities in their Partial Orders ($\prec$): • – Separator; & – Parallelism; | – Alternatives; [...] – Sub-activities
BL	[13]	IEEE SW'02	Pro. Ado.	Scope & Budget Platform $\prec$ Analyze Platform Requirements & Model Variability $\prec$ Design Architecture $\prec$ Design System Model $\prec$ Refine Architecture $\prec$ Design Assets • Analyze Variant Requirements & Select Features $\prec$ Design & Adapt Architecture $\prec$ Adapt Assets & Build Variant
BL	[11]	IEEE SW'02	Pro. Ado.	Develop Assets • Engineer Variant • Manage Platform • Design Architecture • Evaluate Architecture • Analyze Platform Requirements • Integrate Assets • Identify Assets • Test • Configure • Scope Platform • Train Developers • Budget Platform
BL	[14]	UPP'04	Pro. Ado.	Analyze Domain $\prec$ Design Architecture $\prec$ Implement Platform • Analyze Variant Requirements $\prec$ Derive Variant
BL	[12]	Book'05	Pro. Ado.	Scope Platform $\prec$ Analyze Domain [Analyze Commonalities $\prec$ Analyze Variability $\prec$ Model Variability] $\prec$ Design Architecture $\prec$ Implement Platform $\prec$ Test Platform • Analyze Variant Requirements $\prec$ Design Variant $\prec$ Derive Variant [Configure $\prec$ Implement Specifics $\prec$ Build Variant] $\prec$ Test Variant
BL	[15]	Book'13	Pro. Ado.	Analyze Domain [Scope Platform $\prec$ Model Variability] $\prec$ Implement Platform $\prec$ Analyze Variant Requirements $\prec$ Derive Variant
ER	[10]	STTT'15	Ext. Ado.	Analyze Commonality & Variability [Compare Requirements $\prec$ Diff Variants $\prec$ Model Variability] $\prec$ Design Architecture [Extract Architecture $\prec$ Evaluate Architecture $\prec$ Refine Architecture & Variability Model] $\prec$ Develop Assets • Merge Variants $\prec$ Refactor $\prec$ Add Variation Points [Diff Variants $\prec$ Refactor] $\prec$ Model Variability $\prec$ Derive Variant • Analyze Commonality & Variability [Model Variability $\prec$ Compare Requirements & Tests & Diff Variants $\prec$ Refine Variability Model] $\prec$ Extract Platform
ER	[73]	SPLC'16	Ext. Ado.; Vb. Evo.	Diff Variants $\prec$ Analyze Variability $\prec$ Model Variability $\prec$ Add Variation Points $\prec$ Adopt Tooling $\prec$ Compare Requirements $\prec$ Map Artifacts • Develop Assets [Propose Asset $\prec$ Analyze Asset Requirements $\prec$ Design Asset $\prec$ Implement Asset $\prec$ Test Asset] • Release Platform [Plan Release $\prec$ Produce Release Candidate $\prec$ Test Platform] • Release Variant [Scope Variant $\prec$ Derive Variant $\prec$ Test Variant]
ER	[58]	ESE'17	Ext. Ado.	Analyze Commonality & Variability [Locate Features] $\prec$ Model Variability $\prec$ Re-Engineer Artifacts
ER*	[81]	SPLC'17	Ext. Ado.	Diff Variants $\prec$ Locate Features $\prec$ Model Variability $\prec$ Map Artifacts
ER*	[145]	SPLC'18	Ext. Ado.	Model Variability $\prec$ Adopt Tooling • Domain Analysis • Implement Platform • Analyze Variant Requirements • Derive Variant • Configure
ER	[155]	SPLC'18	Ext. Ado.	Train Developers $\prec$ Analyze Domain $\prec$ Model Variability $\prec$ Implement Assets [Analyze Documentation | Diff Variants $\prec$ Refactor]
ER*	[140]	Chapter'19	Ext. Ado.	Analyze Variability $\prec$ Locate Features $\prec$ Map Artifacts
ER*	[53]	ESEC/FSE'19	Ado.; Evo.	Plan Variability Modeling $\prec$ Train Developers $\prec$ Model Variability $\prec$ Assure Quality [Evaluate Model • Test Model]
ER*	[80]	JSS'19	Vb. Evo.	Propose Asset $\prec$ Analyze Asset Requirements $\prec$ Assign Developers $\prec$ Fork Platform $\prec$ Implement Asset $\prec$ Create Pull-Request $\prec$ Review Asset $\prec$ Merge into Test Environment $\prec$ Test Asset $\prec$ Merge into Platform $\prec$ Release Platform
ER*	[95]	SPLC'19	Ext. Ado.; Vb. Evo.	Adapt Variant $\prec$ Propagate Adaptations • Analyze Domain $\prec$ Analyze Variability $\prec$ Locate Features • Extract Platform • Model Variability • Extract Architecture • Refactor • Test Platform • Test Variant
ER*	[46]	ESEC/FSE'20	Vb. Evol.	Scope Variant $\prec$ Design Variant $\prec$ Derive Variant $\prec$ Adapt Variant $\prec$ Assure Quality
ER*	[59]	VaMoS'20	Ext. Ado.	Train Developers $\prec$ Analyze Domain $\prec$ Prepare Variants [Remove Unused Code $\prec$ Translate Comments $\prec$ Analyze Commonality $\prec$ Diff Variants] $\prec$ Analyze Variability $\prec$ Extract Architecture $\prec$ Locate Features $\prec$ Model Variability $\prec$ Extract Platform $\prec$ Assure Quality
SR	[156]	SPLC'15	Vb. Evo.	Scope Variant [Analyze Variant Requirements • Design Variant • Configure] $\prec$ Budget Variant $\prec$ Design & Implement Variant [Analyze Variant Requirements $\prec$ Design & Evaluate Variant $\prec$ Implement & Adapt Variant $\prec$ — | Propagate Adaptations] $\prec$ Configure & Test Variant
SR	[157]	VaMoS'15	Ext. Ado.	Analyze Variability [Diff Variants & Identify Fork Points $\prec$ Classify Adaptations $\prec$ Merge Bug Fixes | [Name Assets $\prec$ Merge Assets into Hierarchy]] $\prec$ Add Variation Points $\prec$ Model Variability $\prec$ Locate Features $\prec$ Extract Platform $\prec$ Configure
SR	[158]	ESE'16	Ado.	Analyze Domain [Gather Information Sources $\prec$ Define Reuse Criteria $\prec$ Collect Information $\prec$ Analyze & Model Variability $\prec$ Extract Architectures $\prec$ Evaluate Results] $\prec$ Budget Platform
SR	[159]	SPLC'16	Pro. Ado.	Engineer Platform [Analyze Platform Requirements $\prec$ Design Architecture & Implement Platform $\prec$ Implement Assets] $\prec$ Derive Variants • Manage Platform
SR	[160]	SPLC'16	Pro./Ext. Ado.	Scope Platform $\prec$ Engineer Platform [Design System Model $\prec$ Design Architecture & Implement Platform $\prec$ Model Variability] $\prec$ Derive Variant [Design Variant [Design Variant Model $\prec$ Scope Variant $\prec$ Select Features] $\prec$ Evaluate Design [Evaluate Design Logic $\prec$ Configure] $\prec$ Design Variant $\prec$ Implement Variant] $\prec$ Test Variant
SR	[161]	VaMoS'16	Pb. Evo.	Analyze Variant Requirements $\prec$ Define Build Rules $\prec$ Configure & Derive Variant $\prec$ Test Variant
SR	[162]	JSS'17	Pro. Ado.	Model Variability $\prec$ Design System Model $\prec$ Derive Variant
SR	[163]	SPLC'17	Vb. Evo.	Fork Platform $\prec$ Test Platform $\prec$ Merge into Platform
SR	[164]	SPLC'17	Pro. Ado.	Design Architecture $\prec$ Add Variation Points $\prec$ Model Variability $\prec$ Configure $\prec$ Derive Variant
SR	[165]	SPLC'17	Vb. Evo.	Derive Variant [Scope Variant $\prec$ Plan Variant [Define Variant Backlog $\prec$ Estimate Efforts $\prec$ Plan Development] $\prec$ Build Variant [Create Backlog $\prec$ Time-Box Control]] • Manage Platform [Scope & Budget Platform]
SR	[166]	SPLC'17	Pro. Ado.	Analyze Platform Requirements [Analyze Domain $\prec$ Scope Platform $\prec$ Model Variability] $\prec$ Design Architecture $\prec$ Evaluate Architecture & Map Artifacts $\prec$ Derive Variant
SR	[167]	ICSE-SEIP'18	Ado.; Pro. Evo.	Analyze Platform Requirements $\prec$ Analyze Commonality & Variability $\prec$ Design Architecture $\prec$ Implement Platform • Analyze Variant Requirements $\prec$ Scope Variant [Identify Assets & Define New Assets] $\prec$ Implement Assets $\prec$ Integrate Assets $\prec$ Configure $\prec$ Test Variant • Map Artifacts • Model Variability • Unify Variability
SR	[168]	SPLC'18	Vb. Evo.	Define Variant Backlog $\prec$ Implement Variant [Analyze Variant Requirements $\prec$ Implement Assets $\prec$ Test Variant] $\prec$ Add Variation Points [Design Variation Points $\prec$ Refactor $\prec$ Test Platform]
SR	[169]	TSE'18	Ado./Evo.	Add Variation Points $\prec$ Adopt Tooling • Manage Knowledge • Resolve Configuration Failures • Assure Quality
SR	[170]	SPE'19	Ext. Ado.	Plan Development [Assign Developers $\prec$ Assign Roles $\prec$ Analyze Documentation] $\prec$ Assemble Process [Select Techniques $\prec$ Adopt Tooling $\prec$ Assign Tasks] $\prec$ Extract Platform [Execute Assembled Process $\prec$ Document Assets $\prec$ Document Process]
SR	[171]	SPLC'19	Pro. Ado.	Analyze Domain [Specify Properties $\prec$ Model Variability • Analyze Variant Requirements [Configure $\prec$ Optimization]] • Derive Variant [Configure $\prec$ Integrate Assets $\prec$ Test Variant] • Implement Platform

BL: BaseLine; ER: Expert Review; SR: Systematic Review
Ext.: Extractive; Pro.: Proactive; Ado.: Adoption; Pb.: Platform-based; Vb.: Variant-based; Evo.: Evolution

contemporary overview of practice, serving as a suitable dataset for adapting the baseline process models. However, we also identified interesting properties of the dataset that were important to consider while constructing promote-pl.
Activities. We unified the terminologies used in the selected publications, and abstracted activities to compare their orders. Still, we kept 99 unique activities, far too many to integrate into promote-pl. There are two reasons for this many activities. First, the publications vary heavily in the level of detail in which they report activities. For example, some simply state "derive product," while others detail single steps of this activity (e.g., "build"). Second, the publications cover various software-engineering methods (e.g., agile, model-driven), domains (e.g., power plants, web services), implementation techniques (e.g., C preprocessor, runtime variability), tools (e.g., fully automated derivation process, build system), and development phases (e.g., business analysis, variant derivation). The varying levels of details and the high diversity mean that it is not possible to unify all terms and activities. We addressed this issue by focusing on re-appearing activities in similar orders.
Partial Orders. As we can see in Table 3.1, we obtained a total of 42 partial orders (without counting sub-orders or alternatives). Interestingly, due to the variations in the activities, there is not a single order that is identical to another order. Still, within a specific scope (e.g., extractive adoption), they share similarities in terms of activities and their orders—while they are quite different between scopes. This indicates again that we require an updated process model for product-line engineering.

Besides the high diversity of activities, one particular reason for the missing overlap seems to be ambiguity of what actions a specific activity comprises. For instance, "analyze domain," "scope platform," and "analyze commonality/variability" are often used together within partial orders. However, their exact orders vary, and sometimes one of these activities is a sub-activity of another. This indicates that it may not be well-understood what activities comprise what concrete actions, for instance, because different process models vary in their definitions. To tackle this problem, we read descriptions in the papers and relied particularly on the descriptions of [12] to reason about design decisions.

3.4.2 Process Model Elements (RO_2)

We display the high-level abstraction of promote-pl in Figure 3.1. The **adoption** includes starting from existing (extractive) or planned (proactive) variants that are integrated into a platform. Alternatively, a planned or existing variant can represent the derived variant that is extended later on (reactive). During the **evolution**, derived variants are evolved to include new features. Such variants can be evolved individually (clone & own) or integrated into the platform by merging features or variants (returning to **adoption**).

We show the detailed representation of promote-pl in Figure 3.3, using a customized representation that builds on UML activity diagrams [175] to ease comprehensibility. The representation comprises nine different elements (summarized in the bottom left corner):

1) *Start Nodes* have essentially the same meaning as in UML, but we allow to start only at one; whereas UML would require to initiate the workflow

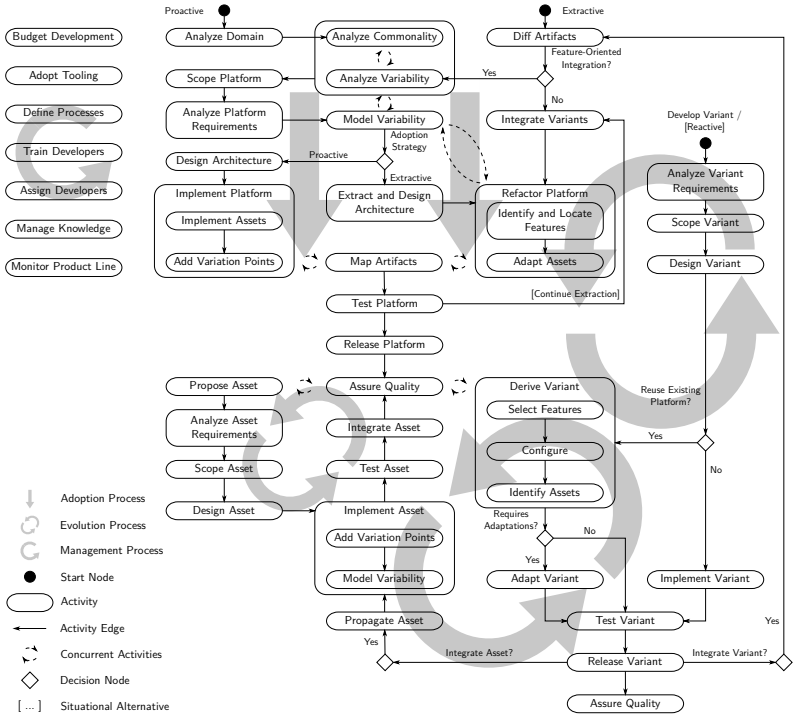


Figure 3.3: The detailed representation of promote-pl.

at all start nodes simultaneously.

2–3) *Activities* and *Activity Edges* have exactly the same meanings and representations as in UML.

4) *Concurrent Activities* are similar to fork and join nodes in UML, indicating that the activities connected by the arrows are (or can be) performed at the same time

5) *Decision Nodes* have the same meaning as in UML, and we explicitly allow that they may have only one outgoing edge; representing an optional workflow.

6) We use *Situational Alternative* to easily represent two scenarios: First, to display that variant development also reflects parts of the reactive adoption strategy. Second, to show that one workflow occurs only if artifacts of variants are extracted (i.e., extractive adoption, evolution via variant integration).

7-9) We abstractly indicate the position of six processes and their workflows in promote-pl, distinguishing three different types. First, *Adoption Processes* (⇓) are the proactive and extractive adoption strategies (as we will explain, reactive adoption represents an evolution process). Second, *Evolution Processes* (↻, ↻) are re-appearing workflows used to extend a product line—usually incorporating forward- and re-engineering activities (i.e., round-trip engineering). Third, the *Management Process* (↻) represents

seven activities that are concerned with enabling and managing other processes, which is why they are performed constantly and in parallel.

We remark that we omit end nodes, since promote-pl reflects adoption and evolution in a round-trip engineering style. So, the end of all processes would mean that the product line is discontinued.

3.4.3 Promote-pl and Adaptations (RO_2, RO_3)

A Different Decomposition. As explained in Section 3.2, organizations appear to decreasingly separate and distinguish domain and application engineering, which is supported by our elicited data and experiences. For organizations, it is more important to understand how to adopt a platform and engineer variants, instead of considering the two phases in isolation (e.g., Danfoss reports platform extraction without fully modeling variability first [73]). Moreover, organizations have mixed teams that employ domain and application engineering in parallel. For example, in some organizations, the same team implements a new variant and refactors it into platform assets, whereas the platform team only tests and quality-assures assets. We reflected this primary concern of interest in promote-pl, moving from domain and application engineering towards the *Adoption* and *Evolution* of a product line. This is a major difference compared to existing process models, and we explain the resulting overarching processes of promote-pl in the following.

Product-Line Management. Some baseline process models comprise activities for managing a product line, often integrated into the domain engineering, but also as a separate phase. Our empirical data suggests that the *management process* (⟳) comprises a challenging and practically important set of activities, enabling organizations to plan and apply product-line engineering successfully. We found that all management activities should run in parallel—to each other and all development activities, which was also suggested before [11].

In particular, we found that seven activities are mentioned as important, for instance, budgeting development activities, adopting tooling as well as processes, and training developers, most of which are mingled and require monitoring of development activities for steering. Interestingly, such management activities have gained less interest in the research community compared to development activities [137]. For instance, budgeting may be supported by cost models, and several of such models have been proposed for product-line engineering. Unfortunately, existing cost models are often limited (e.g., considering their scopes and foundations in empirical data [59,149,176]), and only few experience reports provide guides on how to employ them in practice [158, 177].

Product-Line Aoption. For adopting a product line, we distinguish between the three strategies defined by [132].

First, the *proactive development process* (left ⬇) is identical to the domain engineering of the baseline process models, comprising only minor clarifications. At the beginning, an organization analyzes its domain, comparing its commonalities (which others recommend to start with to identify reuse potential [12]) and variability. Based on the results, the platform is scoped and requirements are derived, which allows to construct a variability model. As we display in Figure 3.3, variability modeling can be performed in parallel to analyzing commonality and variability, as both may affect each other (e.g., refining the variability

model). Considering Figure 3.2, this represents "domain requirements engineering" and "domain design" in the same order, but in a more flexible process. Afterwards, the platform architecture is designed and the platform with its assets as well as variation points is implemented (i.e., "domain implementation" in Figure 3.2). We make two activities more explicit here that have been mentioned multiple times, and thus seem important to include: adding variation points and mapping artifacts (e.g., assets, documentation, variability model) to ensure traceability. Finally, the resulting product-line platform must be tested, released, and quality assured, again fully in line with baseline process models. Overall, this proactive adoption process is close to the domain engineering described by [12]—except for separating management activities and refinements that have been pointed out explicitly in recent publications.

Second, the *extractive development process* (right ⇓) is a first extension to the baseline process models. We actually found different instantiations of this process, both usually starting with diffing of artifacts. On the one hand, an organization can decide to perform a full-fledged feature-oriented integration, meaning that it performs the same analyses (but focused on variability first [53]), scoping, and variability modeling as in the proactive adoption. This represents a top-down approach for extracting the product line. However, after the modeling, an organization usually designs an architecture by extracting and adapting an architecture from the existing variants. Afterwards, the refactoring mainly includes locating the identified features as well as adapting their assets to the architecture and adding variation points.

On the other hand, an organization may decide to simply integrate variants without defining the platform first. Instead, the platform is built by refactoring the integrated variants, which involves identifying and locating features as well as adapting the corresponding assets (e.g., adding variation points, improving re-usability)—representing a bottom-up approach. For managing the product line, the organization must model the refactored variability in parallel. As for the proactive adoption, in both instantiations the organization also has to map artifacts before the platform can be tested and eventually released. However, especially for the second instantiation, the organization may iteratively integrate variants, resulting in a loop.

As we can see, the extractive development process comprises similar activities as the proactive one. Still, there are differences in these activities, for example, in the refactoring of the platform, adapting assets, and the missing domain analysis (i.e., the variants are already established in the domain). Moreover, if an organization does not employ a feature-oriented integration, the process and its order of activities vary considerably.

Third, the *reactive adoption process* is not mentioned in the publications we analyzed. However, this is rather unproblematic, since reactive adoption is only a special case of the *variant-based evolution*. Particularly, a first variant is implemented without the platform, and can afterwards be extended by integrating new assets or variants into the first one. So, promote-pl represents all adoption strategies, and especially for the reactive adoption process we can see that domain and application engineering are mingled.

Product-Line Evolution. The evolution of a product line is driven by new customer requirements. So, while we distinguish between three different evolution processes, they usually start with the development of a new variant,

and the typical application-engineering activities used for requirements analysis and scoping the variant. However, after understanding what new assets are required for developing a variant (i.e., during its design) and deciding to reuse the platform, the individual evolution processes differ.

First, *platform-based evolution* (left ↻) is typically assumed implicitly in baseline process models (cf. Figure 3.2). Thus, the evolution at this point switches from application- to domain-engineering activities. The new asset must be proposed to the platform, designed to fit the platform architecture, implemented, which also includes adding variation points and modeling the new variability, tested, and integrated. To this end, an organization may use feature forks, but the core concept is a fast or continuous integration and close coordination with the platform. Afterwards, the variant can be derived by selecting its features, defining a configuration, and identifying the corresponding assets for integrating them into a repository. Identifying assets can be fully automated based on different technical solutions (e.g., configuration managers), but without such automation developers have to identify and pull the assets from different sources. Finally, the variant may require further adaptations that should not be part of the platform, or can be tested and released as is. Still, we found and experienced that organizations do not employ platform-based evolution, but, instead, rely on the following processes.

Second, during *variant-based evolution using asset propagation* (right ↻), an organization derives and clones a variant from its existing platform that is close to the new variant. In some cases, this clone may even represent the complete platform, for instance, when developing highly innovative variants that may be intended to remain separated. After adapting the variant by adding new assets, the organization may find that these assets are relevant for other variants or even the whole platform. So, assets are propagated to the platform, employing a similar process as for platform-based evolution, namely implementing an asset for reuse, testing its functionality, and finally integrating it into the platform. In particular, we experienced this evolution process for established markets where variants require new assets that have a high potential for various customers, and thus are intended for integration early on. An important prerequisite for this process is that the variant has not co-evolved for too long from the platform, as this challenges asset integration (i.e., the platform may have changed too much for simply propagating the asset)—in which case the third evolution process is more likely.

Third, *variant-based evolution using variant integration* (↻) refers to the re-integration of complete variants into the platform. We found this to be a common case if variants evolved for a longer time without synchronization with the platform, for example, in the case of highly innovative variants, co-evolution resulting in clone&own, or reactive product-line adoption. However, we also found that such variants are re-integrated based on the same process as the extractive adoption: The variant is diffed and then integrated by refactoring it to fit the platform, which may involve variability analysis, scoping, and variability modeling first; or a direct integration and parallel variability model. So, we can see that variant-based evolution, particularly with variant integration, switches the typical order of domain and application engineering, first implementing a variant to then integrate the new assets into the platform.

Domain and Application Engineering. The aforementioned process de-

scriptions already showed that domain- and application-engineering are far more tangled and exist in varying orders compared to baseline process models. So, the typical activities associated with these two phases are represented in promote-pl, but the individual processes iterate between them. Since this is based on the publications we analyzed and aligns with our experiences, it seems that these two phases are rather cross-cutting concerns in contemporary product-line adoption and evolution processes. For this reason, they are still important and helpful to structure product-line engineering, but promote-pl is an important update to provide a more comprehensive, practice-oriented, and recent overview of product-line practices.

By constructing promote-pl, we found:

- A switch in the primary concern of interest from domain and application engineering to adoption and evolution.
- That important management activities must run in parallel to the development, but seem to be less investigated in research.
- That several adaptations to previous process models were necessary to incorporate the three adoption strategies.
- That variant-based evolution via asset- or variant-integration is the major strategy to drive the evolution of a product-line.
- That domain and application engineering are rather cross-cutting instead of primary decomposition concerns.

3.5 SE Practices (RO3)

In this section, we discuss promote-pl's relations to contemporary, trending software-engineering practices, which are typically applied in combination.

Continuous Software Engineering. Referred to as continuous software engineering [135], modern processes increasingly aim at bringing different phases together—reflected in recent practices including continuous integration [178], continuous deployment, continuous testing, or DevOps [179]. This trend is reflected in promote-pl, bringing together domain and application engineering in an iterative, round-trip-like process. The product-line literature recently also emphasized these practices for variability management [163, 167], and we experienced the demand for respective tool and methodological support first-hand with industrial partners [51].

When engineering variant-rich systems, continuous software engineering requires a configurable (product-line) platform. For instance, continuous deployment requires automated configuration, since manually assembling the final system (i.e., variant) cannot be done manually, or using clone&own for frequent (continuous) deployment. Likewise, continuous integration facilitates evolving the trunk using short-lived clones, and continuous testing also requires automated configuration for running test cases.

In this light, *promote-pl resolves a discrepancy between continuous software engineering and the pre-dominant extractive and reactive adoption strategies* [48]

of product lines. It supports adopting a platform extractively or reactively, and evolving the platform via variants. The latter, depending on the extent of architectural deviation from the platform (see the activity *Release Variant* with its decision nodes *Integrate Asset* or *Integrate Variant* in Figure 3.3), can be integrated with the same activities as adopting a platform extractively (deviation) or in a more continuous-integration-like way (no deviation). This aspect of promote-pl unifies evolution and re-engineering, and establishes round-trip engineering.

Clone Management and Incremental Adoption of Platforms. Organizations primarily use clone&own to implement variants [48, 51, 180], which is a cheap and readily available strategy, typically based on using branching facilities in version-control systems [3, 44, 181]. However, the maintenance effort for cloned variants can easily explode. To support evolution [95] before investing in a platform, clone-management frameworks strive to help synchronizing variants and keeping an overview understanding [1, 5, 8, 9, 182, 183]. A step towards clone management are governance strategies for branching and merging [3]—explicit rules for engineers when creating variant branches, aiming at reducing maintenance overhead to some extent. [181], for instance, provide a branching model, which is also instantiated elsewhere [15]. However, with an increasing number of variants, it may still be necessary to adopt a platform. Instead of big-bang efforts, recently, incremental adoption strategies have been proposed [1, 6], aiming at incremental benefits for incremental investments, and therefore avoiding the risks of big-bang migrations, which disrupt development and the ability to sell products [84, 146–148]. Finally, another common practice is to use concepts of a configurable platform (e.g., variation points) together with clone&own [51]. In this light, *promote-pl explicitly supports clone management as well as an incremental adoption of platforms, or using both in a unified manner.*

Dynamic Configuration and Adaptive Systems. Modern, adaptive systems require late and dynamic binding, including microservice [184], cyber-physical [185], industry 4.0 [186], and cloud computing systems [187]. There, resource variations, asset availability, and environmental changes require systems and their software to adapt at runtime. To this end, a platform with variability as well as parameterization mechanisms needs to be adopted. A difference is that such platforms are not necessarily variant-rich systems. Instead, parameterization allows tuning or customizing systems to specific needs at runtime. Not surprisingly, several of our analyzed publications describe product-line engineering in such contexts [80, 155, 156, 161]. In this light, *the adoption strategy is rather reactive, where a single system is developed and gradually extended with variation points, as covered in promote-pl.* Still, better methods and tools are needed to manage and evolve dynamic and adaptive platforms [188, 189].

Agile Practices. Agile software engineering [190, 191] methodologies focus on customer involvement, small increments, and fast feedback. Almost all agile methodologies also build on the notion of features, including SCRUM, XP, and FDD (feature-driven development). They also foster automated testing, which, similar to continuous software engineering, requires configurable platforms.

We found two publications that report to adapt agile methods for their product-line engineering: [73] underpin that product-line and agile engineering are not conflicting, but the developers must be aware that they deliver assets

to a platform that is used by others. Slightly in contrast, [165] find that agile methods are not ideal to *Derive Variants*, because the development cycles are too short to *Train Developers*. However, they adapted agile methods for evolving and throughout multiple product lines, which facilitated their engineering. These experiences indicate that agile methods are important, particularly to *Evolve* a product line (e.g., via its variants). In this light, *promote-pl does not only support agile practices, but is also crucial given the focus of agile methods on features, automation (similar to continuous software engineering), and incremental evolution via variants.*

Simulation in Testing. Three of the analyzed publications, and two of our industrial partners [51], explicitly mention to use simulation environments to *Test* their platforms and variants [51, 73, 160, 161]. While promote-pl captures these activities, more research is needed in this direction. In particular, this is different to sampling variants and test them, as safety-critical systems (e.g., cars, power plants) require an actual simulation environment to test whether software and hardware interact correctly. Moreover, as the simulators may have different properties (e.g., for transferring data) or require additional features (e.g., for *Monitoring* additional data), this can also result in simulation-specific assets (*Add Variation Points*). In this light, *promote-pl covers the relevant activities, and can guide the development of supporting techniques for simulation testing.*

3.6 Threats to Validity

Construct Validity. Regarding construct validity, we may have misinterpreted the terminology used in different publications. Even more, some sets of activities have been used in varying orders in different publications, indicating variations in the use of the constructs we investigated. As a result, the orders of activities we elicited may not completely represent those intended by the authors. We mitigated this threat by building on 33 publications, carefully reading the descriptions of activities in each publication, and reasoning based on our experiences.

Internal Validity. Our work may be threatened by the methodology we employed. We may have falsely disregarded publications, missed important data during the extraction, and not derived the most suitable process model—particularly as we also relied on our experiences and interpretation. However, to limit these threats, we adapted recommendations for process theory [133], suggesting that secondary studies (e.g., systematic literature reviews) are a reliable source for such studies to reduce the potential bias of personal knowledge. Moreover, we were careful to not overly interpret the data we elicited, and checked all outcomes among all authors.

External Validity. The goal of this study was not to derive a universal process theory, which is arguably not possible. Instead, we aimed to capture how product-line engineering is currently done based on empirical data. So, as our data also shows, several process properties, such as the technologies used, the domain of the product line, and the developers involved, limit the transfer of our results to other organizations. We mitigated this external threat by considering various publications and reflecting on our industrial collaborations. For this reason, we argue that we mitigated this threat as far as possible, considering our goal of analyzing current practices.

Conclusion Validity. Other researchers may derive a different process model for product-line engineering, depending on the publications they consider, their experiences, or the construction process. To limit this threat to the conclusion validity, we explained our methodology, reasoned about our modeling decisions, and documented all publications we considered. Thus, we enable researchers to replicate and verify promote-pl.

3.7 Related Work

Software reuse, its methods, technologies, and processes have been studied extensively. Related to our work, [192] survey existing domain-engineering and product-line engineering processes. Similarly, [193] as well as [194] provide general overviews on software reuse, including adoption strategies, methodologies, and techniques. In contrast to us, none of these works derives a process model, and they are rather old—missing insights on current practices and technological advances.

Several researchers, including ourselves, have analyzed how developers reuse software in practice. For instance, we investigated feature-modeling practices [48, 53] in order to understand how feature models are adopted and constructed in practice, but this is only one activity in the process model. Van der Lindern et al. [2] report 10 experience reports of how organizations employ product-line engineering, and what benefits they achieved. However, these cases are comparably old and analyzed in the context of the process model of [12]. In a similar direction [195] compare the reuse practices of two organizations, but do not derive a process model for these. We reviewed other related work, which describes (parts of) product-line engineering processes based on practical experiences, to construct promote-pl (cf. Table 3.1).

Out of the numerous literature studies on product-line engineering [196], the works of [76], [197], and [58] may be the closest to promote-pl. [76] perform a systematic mapping study on product-line evolution, identifying 23 studies on extractive processes. [197] build on that study, including some additional papers based on their selection to derive a taxonomy (which is similar to a process theory [133]) of product-line re-engineering. Most recently, [58] report a systematic literature review, also on re-engineering. In contrast to the other two papers, the authors synthesize a high-level process model (see the corresponding partial order in Table 3.1). All of these works focus on the specific processes of re-engineering product lines, which is part of promote-pl. So, these works are complementary, and they actually argue that well-defined, contemporary process models are needed; which we contribute with promote-pl.

3.8 Conclusion

We presented promote-pl, a modern process model for product-line engineering. Its design is based on a systematic analysis of the literature (experience reports and empirical studies) and our own industrial experiences. We adapted a method for deriving process theories to identify engineering activities and their (partial) orders as reported in the literature, and then unified the terminology to create an aggregated process model. The granularity of promote-pl

allows practitioners to easily map and apply activities to various development processes—less strictly than existing process models.

Core characteristics of promote-pl, as opposed to existing process models, which were conceived almost two decades ago, are the:

- focus on **adoption and evolution strategies** as the dominant decomposition criteria of the process, which is more aligned with primary organizational concerns;
- support for **different adoption strategies**, including the dominant extractive and reactive platform adoptions;
- support to **evolve a platform via its variants** instead of primarily via the platform itself; and
- **alignment with modern practices** including continuous software engineering, agile methods, clone management, incremental platform adoption, and simulation-based testing.

We envision that future research will investigate the adoption of promote-pl in case studies, and build corresponding tool support. Also, we hope to inspire practitioners providing experience reports and requirements for tools supporting promote-pl.

Chapter 4

Paper C

Effects of Variability in Models: A Family of Experiments

Wardah Mahmood, Daniel Strüber, Anthony Anjorin, and Thorsten Berger

Accepted In Empirical Software Engineering Journal (2022).

Abstract

The ever-growing need for customization creates a need to maintain software systems in many different variants. To avoid having to maintain different copies of the same model, developers of modeling languages and tools have recently started to provide implementation techniques for such variant-rich systems, notably *variability mechanisms*, which support implementing the differences between model variants. Available mechanisms either follow the *annotative* or the *compositional* paradigm, each of which have dedicated benefits and drawbacks. Currently, language and tool designers select the used variability mechanism often solely based on intuition. A better empirical understanding of the comprehension of variability mechanisms would help them in improving support for effective modeling.

In this article, we present an empirical assessment of annotative and compositional variability mechanisms for three popular types of models. We report and discuss findings from a family of three experiments with 164 participants in total, in which we studied the impact of different variability mechanisms during model comprehension tasks. We experimented with three model types commonly found in modeling languages: class diagrams, state machine diagrams, and activity diagrams. We find that, in two out of three experiments, annotative technique lead to better developer performance. Use of the compositional mechanism correlated with impaired performance. For all three considered tasks, the annotative mechanism was preferred over the compositional one in all experiments. We present actionable recommendations concerning support of flexible, tasks-specific solutions, and the transfer of established best practices from the code domain to models.

Keywords: variability mechanisms, model-driven engineering, software product line engineering, empirical study

4.1 Introduction

Variant-rich systems can offer companies major strategic advantages, such as the ability to deliver tailor-made software products to their customers. Still, when developing a variant-rich system, severe challenges may arise during maintenance, evolution, and analysis, especially when variants are developed in the naive *clone-and-own* approach, that is, by copying and modifying them [12]. The typical solution to these challenges is to manage variability by using dedicated *variability representations*, capturing the differences between the variants [2]. An important type of variability representation are *variability mechanisms*, which are used to avoid duplication and to promote reuse when implementing variability in assets such as code, models, and requirements documents. Over more than three decades, researchers have developed a plethora of variability mechanisms, albeit mostly for source code [15, 198, 199].

As companies begin to streamline their development workflows for building variant-rich systems, they recognize a need for variability management in all key development artifacts, including models. The use of models is manifold, ranging from sketches of the system design, to system blueprints used for verification and code generation. The car industry is particularly outspoken on their need for model-level variability mechanisms [200]. For example, General Motors named support for variation in UML models as a major requirement [201], and Volkswagen reported large numbers of complex, cloned variants of Simulink models in their projects [202]. Beyond automotive, the need for model-level variability has been documented for power electronics, aerospace, railway technology, traffic control, imaging, and chip modeling [51].

Recognizing this need, researchers have started building variability mechanisms for models. Variability mechanisms are now available both for UML [18, 20, 203] and Domain-Specific Modeling Languages (DSMLs* [19, 21, 22, 36, 204–208]). Building on these results, researchers have started to address advanced problems such as the migration of a set of "cloned-and-owned" model variants to a given mechanism [108, 202, 209–211], and efficient analysis of large sets of model variants [212–214]. Adoption in several industrial DSMLs has demonstrated the general feasibility of model-level variability mechanisms in practice [215].

While variability mechanisms for source code are reasonably well understood [25, 198, 199], language and tool designers are offered little guidance on selecting the most effective variability mechanism for their purposes. In fact, there is a lack of evidence to support the preference of one mechanism over the other. In line with previous studies on code-level mechanisms [17, 25, 216, 217], we argue that *comprehensibility* is a decisive factor for the efficiency of a variability mechanism—for any maintenance and evolution activity (e.g. bugfixing, feature implementations), the developers first need to understand the existing system. A better empirical understanding of the comprehension of variability mechanisms could support the development of more effective modeling languages and tools.

To this end, we present an empirical study of variability representations in models. We report on a family of three experiments in which we studied how the choice of variability mechanism affects performance during model

*DSMLs allow modeling software systems from different domains using domain-specific notations.

comprehension tasks. We consider comprehension tasks for three popular model types[†]: class diagrams, state machine diagrams, and activity diagrams. The experiments are fully randomized, and employ student developers from three countries. We consider two selected variability mechanisms that are representative for the two main types distinguished in the literature [218]: *Annotative* mechanisms maintain an integrated, annotated representation of all variants. Examples include preprocessor macros [219] (for code) and model templates [203] (for models). Annotative mechanisms are conceptually simple, but can impair understandability, since they clutter model or code elements with variability information [25, 219]. *Compositional* mechanisms allow to compose a set of smaller sub-models to form a larger model. Examples include *feature-oriented programming* [220] (for code) and *model refinement* [221] (for models). Compositional mechanisms are appealing, as they establish a clear separation of concerns, but they involve a composition step which might be cognitively challenging. We aimed to shed light on the impact of these inherent trade-offs.

We focus on three model types used in various modeling languages: class diagrams, state machine diagrams, and activity diagrams. The diagrams are three commonly used types of UML models, popular both in academia and industry. *Class diagrams* play a significant role in domain and system analysis and design. They are representative for a wide array of visual languages modeling domain concepts, such as Entity-Relationship diagrams (ER diagrams, [222]), and they can be used for generating the architecture of a system [223]. *State machine diagrams* model system behavior in terms of the different states a system exists in. They play an important role in software verification [224]. *Activity diagrams* also model behavior, but in contrast to state machine diagrams, they model the interaction between the user and the system. Both state machine diagrams and activity diagrams are representative of other behavioral representations such as sequence diagrams (which model a system in terms of sequential interactions between actors), and can also be used for code generation and system verification [225].

We make the following contributions:

- We present our findings on a family of experiments, each investigating how the choice of variability mechanism affects the comprehensibility of different model-related tasks for three popular model types.
- We present a quantitative analysis of correctness, the completion time, and subjective assessments of our participants for six model comprehension tasks.
- We present a qualitative analysis of participant responses, adding rationale to explain the observed results.
- Based on our synthesized findings, we propose recommendations for language and tools developers.
- We provide a replication package [226] that includes our experimental material, anonymized responses, and analysis scripts.

[†]In this paper, we use the terms *diagram* and *model* interchangeably. In modeling languages such as UML, models can consist of a single diagram. The difference between such a model and the contained diagram is then not essential.

This paper considerably extends our earlier conference paper [89] that presented the first of our three experiments, focusing on class diagrams (73 participants). Based on these earlier results, we designed, performed, and analyzed two follow-up experiments, keeping the methodological setup of the first experiment, while varying the considered modeling languages. The results add nuance and additional insights, while largely confirming the findings from the first experiment. Based on the results, we are able to derive conclusion about a broader class of modeling languages and based on more observations (from 164 participants).

We present the first empirical study of variability mechanisms for models that investigates the effect of different mechanisms in a controlled experiment. While earlier empirical studies considered the comprehensibility of code-level variability mechanisms (see Section 4.8), their generalizability to models is unclear. Code usually has a tree-like structure and is expressed in textual notations. Modeling languages support the structuring of models in a graph-like manner and usually have graphical notations. Since different representations are known to affect performance during decision-making tasks [227], specifically, software engineering tasks [33], we argue that the comprehensibility of model variability mechanisms requires a dedicated investigation. In the scope of models, related work is on experience reports in variability modeling (e.g., [21, 142, 143]) and controlled experiments outside the scope of variability (e.g., [34, 35, 228]).

4.2 Background

There has been a recent surge of interest in dedicated variability mechanisms for models. Lifting the related distinction from code-level mechanisms, two main types are distinguished: Annotative mechanisms represent variability with an annotated integrated representation of all variants. Mechanisms in this category are *model templates* [19, 112, 203], *union models* [229], and the *top-down approach* [230]. Compositional mechanisms represent variability by composing variants from smaller sub-models (from here referred to as *model fragments*). Available approaches mostly differ in their model fragment syntax and composition semantics. Examples are *delta modeling* [231], *model superimposition* [18], *refinement* [208, 221], *components* [36], and the *bottom-up approach* [230].

To illustrate the role of both types of mechanisms in industry, we refer to a recent survey of variability support in 23 DSMLs [215]. It describes four strategies being used: First, a model represents one variant (9 languages); second, elements are reused across models by referencing (10 languages); third, multi-level modeling is used for capturing variability (1 language); fourth, elements have so-called presence conditions (explained shortly, 3 languages). The first strategy is considered as a baseline in our experiments. The second and third one are compositional, as they spread differences between variants across several smaller models. The fourth one is annotative.

We selected the two variability mechanisms for our experiments based on the following criteria: (M1) *The mechanism has a graphical syntax.* (M2) *The mechanism is supported by available tools.* (M3) *The mechanism has been de-*

scribed in the scientific literature. The rationale for M1 was to study variability mechanisms in the widespread graphical representation of our considered model types. Support by tools (M2) and available literature (M3) may contribute to the transfer of existing research results to industrial practice, and allow practitioners to test the mechanisms in available prototypes.

Based on these criteria, as an annotative mechanism, we identified *model templates* (implemented in FeatureMapper [19], SuperMod [20], and Henshin [205]). For compositional, we identified two existing approaches fulfilling the criteria: Delta modeling (implemented by DeltaEcore [232] and SiPL [233]) and model refinements (implemented by eMoflon [22]). We decided to consider model refinements, as they implement the compositional paradigm in the most straightforward way (delta modeling supports deletions, which increases its expressiveness, but requires more complex syntax and semantics).

Example. We illustrate the specific variability mechanisms used in our experiments with a simple example, inspired by Schaefer [234]. The same example was also used in the experiment to introduce the variability mechanisms to the participants.

The example represents a simple cash desk system that exists in three similar, but different variants. Figure 4.1 depicts the individual variants using separate class diagrams: Variant `var1` consists of a `CashDesk` with a `KeyBoard` and a `Display`. Variant `var2` has additionally exactly one `CardReader` connected to the `CashDesk`. Variant `var3` replaces the `Keyboard` with a `Scanner` and makes the `CardReader` optional (multiplicity `0..1` instead of `1`).

The depicted representation of listing variants individually is used as a baseline in the first of our experiments, referred to as the "enumerative mechanism." This solution is frequently applied in practice [215], where it leads to severe maintenance drawbacks. For example, a bug found in one of the variants must be fixed in all variants separately. The goal of the variability mechanisms presented below is to simplify working with such similar, but distinct variants.

Annotative Variability. The annotative mechanism considered in our experiments is *model templates* [203]. Like annotative mechanisms in general, it combines all model variants into a single model with annotations. The left-hand side of Figure 4.2 shows a model template for our example: a class diagram that represents the three variants of the cash desk system. Parts of the class diagram are annotated with *presence conditions*, stating the variants in which the part occurs. For brevity, we define a presence condition as a list of configuration options (disjunction). For example, the presence condition «`var1,var2`» indicates that the annotated part is present when either the configuration option `var1` or `var2` is selected. The absence of a presence condition denotes that the part is contained in all variants.

Colors are used in the following way: Elements (classes and associations)

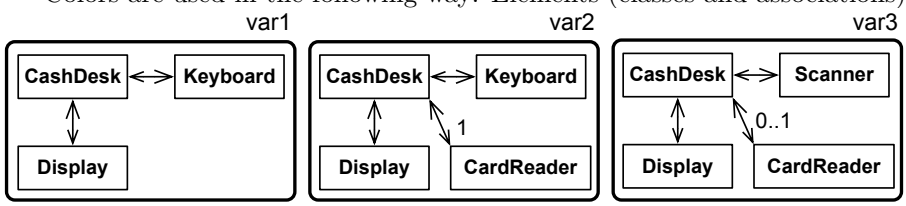


Figure 4.1: Three variants of a cash desk system

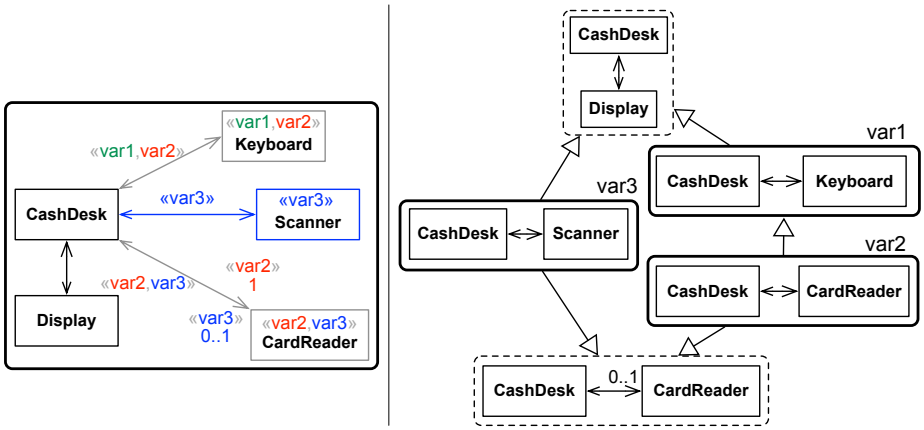


Figure 4.2: Annotative and compositional variability

with a black outline occur in all variants, elements with a grey outline occur in two or more variants, elements with a colored outline belong to precisely one variant, whose annotation is also depicted with the same color. The use of colors to distinguish elements goes back to the original paper that introduced model templates [203]. Colors may be crucial for comprehensibility. In the case of code-level variability mechanisms, Siegmund et al. [216] found that colors support understanding of annotative variability. We are interested to determine if this finding also applies to models.

Individual variants are derived from the combined model as follows: The user sets one of the configuration options as active. The concrete model is derived by removing all those elements whose presence condition does not contain the configuration option. For example, selecting the configuration option `var1` leads to the model variant `var1` in Figure 4.1.

Compositional Variability. The compositional mechanism we considered is *refinement* [22]. Like all compositional mechanisms, refinement provides (i) a means of decomposing variants into smaller building blocks, and (ii) a means of *merging* building blocks to form complete variants. This allows for sharing and reuse of common parts in different variants. The building blocks are visually shown as a network, as depicted in the right-hand side of Figure 4.2. Commonalities of `var1` and `var3`, as well as `var2` and `var3` have been extracted into separate "super" class diagrams. These diagrams have a dashed border, as they only represent commonalities and are "abstract" in the sense that they are not complete variants. Composition of diagrams is denoted using an inheritance arrow, e.g., `var2` is formed by combining `var1`, the elements specified in `var2`, and the elements in the common super class of `var3` and `var2`. As the example demonstrates, multiple super class diagrams (see `var3`) and transitive composition (see `var2`) are possible.

Deriving individual variants is a two-step process. First, a *union* of the contents of the variant and all its transitive parents is computed; this results in a single, flat class diagram (with no parents). Second, a *merge* operator is used to combine elements that should be the same. For class diagrams, this operator combines all elements with the same name. The merge operator also defines how to resolve conflicts: for class diagrams, a common subtype must exist for nodes to be merged, and multiplicities of merged edges are combined

by taking the maximum of lower bounds and minimum of upper bounds. For example, when the variant `var1` is selected, it is merged with its parent (top class diagram with dashed lines). Building the union of both class diagrams and merging the cash desk elements leads to the model variant `var1` in Figure 4.1.

Considered model types. In our experiment family, we cover three types of models: class diagrams, state machine diagrams, and activity diagrams. Our rationale is three-fold: These model types: (i) are commonly taught in undergraduate education, allowing our participants to work with languages they are already familiar with; (ii) are widely used in industry [235, 236], indicating their representativeness, (iii) together are suited to capture three essentially different concerns: static structure, dynamic behavior from the internal system perspective, and dynamic behavior in interaction with the user.

In each of these model types, variability can be implemented using the annotative and compositional mechanisms illustrated in the example above. We now discuss each model type and any necessary customizations required to accommodate the model type in our considered mechanisms.

Class diagrams have a pronounced role in system design and analysis. In code generation contexts, they are used to generate data management components (e.g., large parts of enterprise web and mobile apps can be generated from class models [237–239]), object-oriented code in roundtrip engineering scenarios [240], and ample Model-Driven Engineering (MDE) tooling in modeling platforms, such as the Eclipse Modeling Framework (EMF [241]). Class models can be supported via model templates and model refinement in a straightforward day, as shown in the example above.

State machine diagrams capture the dynamic behavior of a system, focusing on one of its entities or objects, in terms of its possible states and the transitions between the states, based on certain well-defined events. Like class diagrams, they play an important role in code generation [242, 243]. Expressing variability in state machine diagrams using model templates is straightforward, based on assigning presence conditions to states and transitions. Model refinement can be applied in a similar way as for class diagrams. One additional complication arises with hierarchical state machines, where merging must respect the nesting of states as defined in different model fragments. In principle, multiple model fragments can make conflicting contributions to the merge result (for example, if there is a fragment with state A nesting a substate B, and another fragment with a state B nesting a substate A). We designed our examples to avoid such situations.

Activity diagrams, like state machine diagrams, also capture the dynamic behavior of a software system. However, in contrast to state machine diagrams, activity diagrams provide means to model *user* and *user-visible* activities, as well as the flow between them. Activity diagrams allow modeling overlapping activities (*fork*), or activities that need coordination (*merge*). Expressing variability in activity diagrams using model templates is straightforward [244], based on assigning presence conditions to activities. Model refinement can be applied in a similar way as before. One complication concerns the merging of flow: if the same activity appears in different fragments with different subsequent activities, one needs to define how the arising conflict is resolved. For our experiments, we defined a conflict-resolution rule that an actual activity

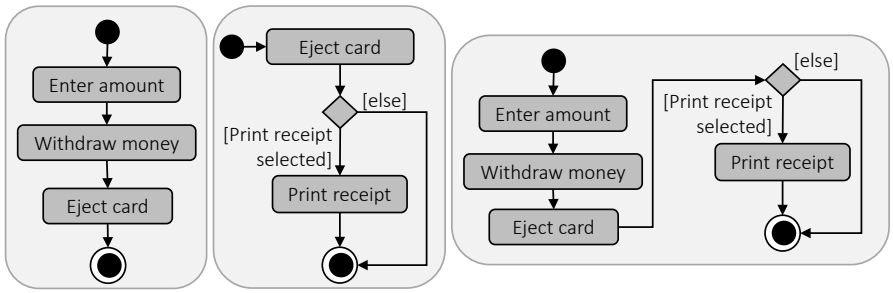


Figure 4.3: Model refinement rule for activity diagrams: ending activities are overwritten by decision nodes

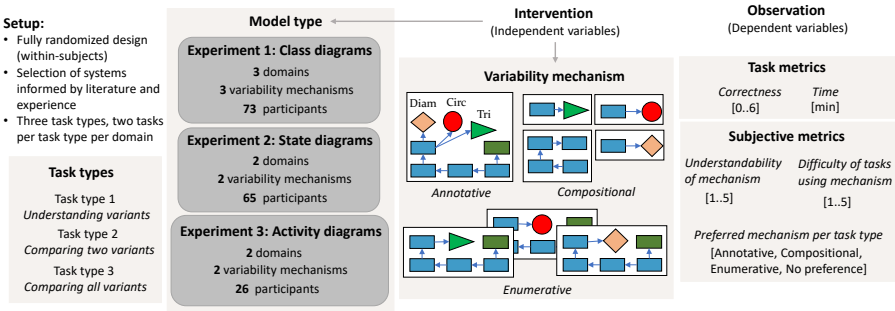


Figure 4.4: Methodology overview

always overrides an ending node. Figure 4.3 depicts an example: the left fragment consists of the functionality of *withdraw* and the middle fragment composes the activities for *printing receipts*. When merging both fragments, the final activity in the left fragment is overwritten by the decision node (represented as a diamond) in the middle fragment. The right fragment shows the merged form of both fragments.

4.3 Overview on Our Family of Experiments

We performed a family of three experiments, illustrated in the high-level overview in Figure 4.4. Our experiment family consisted of two independent variables: the considered model type and variability mechanism. The former was varied *between* experiments, i.e., each experiment focused on a single model type. The latter was varied *within* each experiment, i.e., each experiment compared multiple variability mechanisms on the same model type.

We considered three variability mechanisms—*annotative*, *compositional*, and *enumerative*—where the enumerative mechanism (a simple listing of all variants) was considered the baseline. In all three experiments, we adopted a within-subject design, where each participant used each considered variability mechanism on all tasks. We considered three widely used models types—class diagrams, state machine diagrams, and activity diagrams—as discussed and motivated in Section 4.2.

In Experiment 1, we considered all three variability mechanisms. Based on some important observations from the experiment 1 (explained shortly),

we decided to not consider the enumerative mechanism in the remaining experiments. This allowed us to have a more in-depth comparison and reflection of annotative and compositional variability in experiments 2 and 3, based on more intricate tasks.

In every experiment, our participants worked with example models from certain *domains*, derived from the literature and our experiences. To avoid learning effects based on answers of previously completed tasks, we varied the considered example domain for each variability mechanism. Consequently, the number of considered domains in each experiment matched the number of variability mechanisms: three in Experiment 1, two in Experiment 2 and Experiment 3.

Prior to the experiments, as a preparatory study, we conducted a further experiment (explained shortly in Section 4.4) to shape the design of our materials and tasks. The goal was to assess the suitability of our experimental tasks and to derive potential improvements of the setup.

To recruit a significant number of participants, we involved students as participants, due to their representativeness as stand-ins for practitioners [245]. The participants came from four different universities in Germany (two universities), Netherlands, and Sweden. We shortly discuss demographic aspects of our participants. In each experiment, we randomly divided participants into n groups (n = number of variability mechanisms). For subject allocation, we used a *Latin square design* [25, 26] to ensure that each participant used every distinct variability mechanism and domain *exactly* once, as to avoid learning effects.

To ensure homogeneity, we kept other aspects constant across our experiments as far as possible: training material, goals and research questions, experimental design, task types, task metrics, subjective assessments, analysis, and participant selection. We used different domains in each experiment, deliberately to make the results generalizable. We customized the individual tasks according to the domains, keeping the task types unchanged. To analyze comprehensibility, we designed three task types: Understanding variants, comparing two variants, and comparing all variants. Our detailed methodology and analysis is described in Section 4.5 and Section 4.6, respectively.

The considered domains were varied between the experiments: In Experiment 1 and Experiment 3, we chose intuitively understandable examples that inherently lend themselves towards being expressed with the considered model type. To this end, we derived the domains by taking inspiration from literature. The domain choice in Experiment 3 also incorporated participant feedback from previous iterations of the experiment. In Experiment 2, we derived two sub-domains from a software project considered in the course that the participants were recruited from. This was useful because it allowed us to conclude that all participants were familiar with the considered domains.

4.4 Preparatory Study

We conducted a preparatory study to evaluate our experimental design and identify possible issues and other amendable aspects. The study was performed on a population of 28 students (disjoint from the population of our experiment). The students were familiar with class models, the model type used in the

experiment.

The tasks considered were *bug-finding tasks*, a typical task type for assessing the usefulness of visual representations [25, 95]. Participants were handed a textual requirement specification, together with design models implementing the requirements with one of the given variability mechanisms. The design models contained a number of deviations from the textual requirements (bugs), which the participants were asked to identify. We also asked the participants to suggest potential improvements to the experiment using an open-ended question.

To provide meaningful example domains, we considered the existing literature. The first example represented a phone product line with phones being conditionally capable of making incoming and outgoing calls [18]. The second example represented a project management system with managers, employees, and tasks [246]. Students obtained a virtual instruction sheet and a link to an explanation video for the used variability mechanisms. The students were asked to complete the entire questionnaire in 30 minutes.

From this preparatory study we drew three main conclusions: First, example models with 3 to 4 classes, and 3 or 4 variants each were too simple to demonstrate a difference between both mechanisms. This conjuncture was supported by one participant's written recommendation to *"create [more] complicated examples with 6 or 7 classes and not so easy ones."*

Second, despite our efforts to provide clear requirements, a participant asked us to be "more specific and less ambiguous with the requirement specifications." Ambiguity is an inherent risk to experimental validity since its effect is hard to quantify (it is unclear how many participants assume a different understanding than intended). Another, recurrent comment was that reading the descriptions was tiring, threatening the completion rate. Therefore, we decided to switch the nature of the used tasks in the main experiment to comprehension tasks that do not rely on additional artifacts.

Third, the provided instruction video was viewed as redundant, as it showed only information that was available on the instruction sheet. In the actual experiments, we decided to omit the instruction video.

4.5 Methodology

In this section, we present the detailed experimental methodology for our family of three experiments, each of which is focusing on one model type. Our experimental materials and data, including the raw data, are publicly available via our replication package [226].

4.5.1 Experimental Setup

As explained in Section 4.3, to ensure uniformity, we reused some components of our experiment in all three executions: goals and research questions, training material, task types, task metrics, subjective assessment, quantitative feedback, and data analysis. In this section, we elaborate these common components of our experiments.

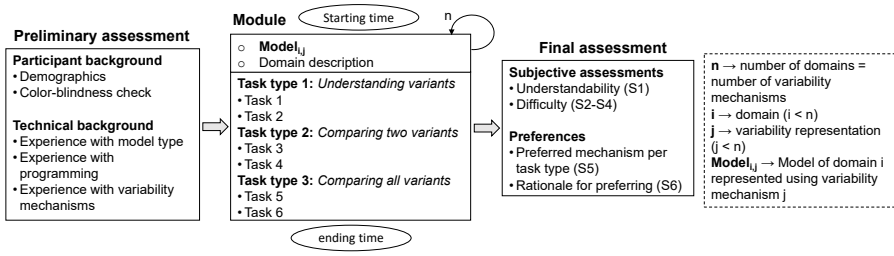


Figure 4.5: Questionnaire design for our experiments, with n=3 for experiment 1, and n=2 for experiments 2 and 3.

Research Questions. Our goal was to study the effect of variability mechanisms on model comprehension. Towards this goal, we formulated and investigated the following research questions:

> **RQ1** *To what extent do variability mechanisms impact the efficiency of model comprehension?*

We studied the effect of annotative and compositional mechanisms on the ability to solve model comprehension tasks correctly and quickly.

> **RQ2** *How are variability mechanisms perceived during comprehension tasks?*

We studied the perceived understandability and difficulty to complete model comprehension tasks depending on the used variability compositional mechanism, based on subjective assessments.

> **RQ3** *How are participant preferences for variability mechanisms distributed over different task types?*

We elicited qualitative and quantitative data about the participants' subjective preferences by asking them to choose a preferred mechanism and explain the choice.

Experimental Design. We applied a *cross-over trial*, a variant of the within-subject design [247], in which all participants are sequentially exposed to each treatment. The treatments in our case were the use of the different variability mechanisms during comprehension tasks. The main benefit of the chosen design is its efficiency in enhancing statistically valid conclusions for a given number of participants. The design also reduces the influence of confounding factors, such as participant expertise, because each participant serves as their own control.

A main threat to this kind of study design are learning effects: during the experiments, participants might transfer experience gained by solving one task to other tasks. We mitigated this threat by using a *Latin square design* [25,26]. Participants were randomly distributed across equally sized groups, such that each participant experimented with each variability mechanism and each domain once. Each group was assigned one of several *paths* through the experiment, based on the different possible combinations of domains and variability mechanisms. For example, consider the three paths for Experiment 1, which included the variability mechanisms annotative (*Ann*), compositional (*Com*), and enumerative (*Enu*), and three domains *d1*, *d2*, and *d3*:

- *Enu d1*→*Ann d2*→*Com d3* (path 1),

- *Com d1→Enu d2→Ann d3* (path 2), and
- *Ann d1→Com d2→Enu d3* (path 3).

Following our Latin square design, to avoid bias related to the complexity of the considered domains, the order of domains was fixed between paths.

Figure 4.5 shows the design and flow of our questionnaire. Each module corresponded to one element of the above-mentioned paths, and consisted of one model, its description, and six tasks. We discuss further threats and mitigation strategies in Section 4.7.

Training Material. The participants received training material in the form of handouts elaborating the variability mechanisms before beginning the experiments (available in replication package [226], *intro* documents in folder *material*). In the training material, we elaborated the variability mechanisms and presented different representations of the illustrative example shown in Section 4.2. We also showed how variants can be derived by giving valid feature selections (annotative) or composing different sub-models (compositional). We reused the same training material across all experiments, with small adjustments. Specifically, we extended the material for Experiment 3 (activity diagrams), where the composition required an extra step, i.e., when merging two activity diagrams, decision nodes overwrite ending activities.

Preliminary Assessment. In the questionnaires (available in replication package [226], folder *questionnaires*), before the actual tasks, we asked the students to self-assess their expertise in three relevant categories using five-point Likert scales: Model type (the baseline technology of our experiment), programming (to argue for the representatives of our findings), and the considered variability mechanisms (the experimental treatment). The five-point scale consisted of positive integers from 1–5, 1 representing the lowest value, and 5 representing the highest. Table 4.1 shows a summary of participant self-assessment ratings for the three experiments.

Task types and Tasks. While designing tasks, we aimed for representativeness. We designed task types to depict common activities performed by developers in variant-rich systems. For each domain represented using a given variability mechanism, participants were required to perform tasks of *three* types. Every task type consisted of *two* concrete tasks. Below, we discuss the three task types, provide an argument for their representativeness, and give examples for the concrete tasks per task type.

Task Type 1 ("understanding variants") required participants to map *elements* of the model (classes, states or activities) to variants. This task type is inspired by feature location: a common activity where parts of an artifact (code or model) implementing a feature has to be identified. Tasks of this type followed the style: *"Which variants have the elements X and Y? List all such variants, or write none otherwise."* Consider the Phone management domain in the first experiment (in the replication package [226], *material/exp1/Ship.pdf*), we formulated the following task for task type 1: *How many variants have both the classes "Camera" and "Video"?*

Task Type 2 ("comparing two variants") required participants to differentiate two variants in terms of the elements they consist of. Specifically, they needed to list the non-overlapping elements of two given variants. In practice, such a task is performed to deeply understanding how two closely related variants differ.

Tasks of this type followed the style: *"How do the two variants Var1 and Var2 differ? List all differing elements if there are any.'* As an example, consider the Robocode domain in the second experiment (see the replication package [226], *material/exp2/Boat.pdf*), where we formulated the following task for task type 2: *Which state(s) does [the robot variant] "Grizzlyman" have that [variant] "Toxitonic" doesn't have? List all such states if there are more than one.*

Task Type 3 ("comparing all variants") involved a broader comparison, e.g., listing elements belonging to all variants. Such a task type is performed when trying to understand the entire variant space. Tasks of this type resembled the following style: *"Which elements are included in all variants?"* To reduce effort in the case of larger examples, we specified a pre-selection of a obvious elements and asked the participants to fill in the remaining ones. For example, consider the third experiment with activity diagrams as the featured model type (replication package [226], *material/exp3/Train.pdf*). We formulated the following task for task type 3: *What is the longest possible path of activities you can have in any of the products, from a start to an end activity? Please list all activities that the path consists of (excluding start and end activities) in order. If there are multiple longest paths, pick one.*

Task types in the questionnaire were represented as sub-sections of the modules (Figure 4.5), where each sub-section consisted of two tasks. Each task consisted of one concrete question that the participants were asked to answer, as illustrated in the previous examples.

Dependent Variables. For measuring efficiency (RQ1), we elicited two metrics: correctness and completion time. The correctness of a task type was the aggregate correctness of its two concrete tasks. For each task, we evaluated the responses using a scale of 0–1 as follows: Correct responses received a score of 1, partially correct 0.5, and incorrect 0. The responses were evaluated against *oracles* that the authors produced. For each task type per sub-system, the primary and secondary authors iteratively solved each individual task until they reached a consensus (which formed the oracle). In majority of the cases, both the authors had the same responses. As a margin of error, we also checked if the responses to one task were consistently different than our oracles. This was never the case in our experiments.

A response was deemed partially correct if it included *some but not all* correct elements, or *some correct and some incorrect* ones. The scores of both tasks of a task type were summed up to obtain the correctness score in the range 0–2. This resulted in a total of three correctness scores per participant. For completion time, we asked the participants to log the starting and ending times of each module (Figure 4.5), which we converted to the completion time in minutes.

To address RQ2, we asked the participants to assess the *understandability* of each mechanism and the *difficulty* of addressing each task type using each mechanism. Specifically, we asked the following questions:

(S1) How easy did you find it to understand each mechanism?

(S2) How difficult was it to answer the questions on "Understanding variants" (tasks 1 and 2) for each mechanism?

(S3) How difficult was it to answer the questions on "Comparing two variants" (tasks 3 and 4) for each mechanism?

(S4) How difficult was it to answer the questions on "Comparing all variants"

(tasks 5 and 6) for each mechanism?

Following the common practice for subjective responses, we captured the response on a 5-point Likert scale for each mechanism. The points represented increasing levels of difficulty (from 1 easiest to 5 hardest). We used the same labels for the Likert scales for S1-S4, specifying explicitly for each question that 1 means *very easy*, and 5 means *very difficult*.

We aimed to investigate two hypotheses: that (i) all variability mechanisms are perceived to be equally understandable, and (ii) while performing tasks of each task type, participants experienced equal difficulty using all variability mechanisms. For the former, we conducted a statistical analysis to compare the understandability of different variability mechanisms included in the experiment. For the latter, we conducted dedicated statistical analyses to compare difficulty ratings per variability mechanism given by the participants for each task type.

For RQ3, we asked the participants to specify their *preferred mechanism* per task type. To gain deeper insight into the rationale, and complement the quantitative information with qualitative data, we also asked our participants to elaborate on their choice of preferred mechanism—a setup inspired by mixed-method research [248]. We used the following questions:

(S5) Which mechanism do you prefer for each of the three task types?

(S6) Can you explain your subjective preferences (intuitively)?

The answer to S5 was specified by selecting one of the literals Annotative, Compositional, Enumerative, None for each of the task types. To collect the qualitative data in S6, we kept S6 open-ended.

Analysis. For hypothesis testing, we used the Wilcoxon signed-rank test [249] which we applied to the task and subjective metrics, following recommendations according to which this test can in fact be applied to Likert-scale data [250]. We used the standard significance threshold of 0.05. Two measurements involved multiple comparisons (correctness and difficulty; each for 3 different task types). For these metrics, we applied the Bonferroni correction [251], yielding a corrected significance threshold of 0.017, obtained by dividing 0.05 by 3. We employed the A_{12} score for assessing effect size following Vargha and Delaney's original three interpretations [41]: $A_{12} \approx 0.56 = small$; $A_{12} \approx 0.64 = medium$; and $A_{12} \approx 0.71 = large$. All tests were executed with R.

For assessing the qualitative data, one of the authors used *inductive coding* to tag the participants' comments from one of the experiments with relevant keywords. Afterwards, two other authors verified the tags and suggested improvements in the tags. Based on the discussion and feedback with the two

Table 4.1: Technical background of our participants

Exp.	Experience with				
	MType	**Prog.**	**Ann.**	**Com.**	**Enu.**
1	3.47±0.60	3.62±0.74	1.73±0.87	1.86±1.03	1.87±0.93
2	2.84±0.58	3.42±0.74	2.26±0.73	2.52±0.77	-
3	3.39±0.64	4.0±0.46	2.1±0.8	2.3±0.86	-

Ratings on a 5-point Likert scale. 1: lowest 5: highest.
Exp.: Experiment **MType:** Model type **Prog:** Programming **Ann:** Annotative Variability **Com:** Compositional Variability **Enu:** Enumerative Variability

authors, the author tagged the comments for the remaining two experiments. The tags were useful to identify interesting aspects, and their frequency helped to identify the redundant concerns.

Participants. The participants were recruited from undergraduate and graduate courses at various universities. Our rationale for recruiting students is their suitability as stand-ins for practitioners: students can perform involving unfamiliar software engineering tools equally well as practitioners [245]. The students were recruited from courses with completed previous lectures and homework assignments on models featuring in the experiment. Before the experiment, it was pointed out that participation in the experiment was entirely voluntary, and data would be stored anonymously. To encourage participation, a gift card raffle was offered as a prize to interested participants.

4.5.2 Experiments

We now discuss the individual aspects of our experiments: model types, domains, participant demographics, and application of the Latin square design.

Experiment 1 was focused on *class diagrams*. In this experiment, in addition to the annotative and compositional mechanisms considered in all experiments, we included the enumerative mechanism as a baseline for comparison.

To select the systems, we specified a set of criteria that a subject system would need to fulfill: (*C1*) The system has been introduced in previous literature. (*C2*) The system comprises several variants. (*C3*) The system has not been introduced in a context related to a particular variability mechanism. The rationale of these criteria was to select systems that represent real variability, rather than making up artificial examples on the spot. As a result, we derived variability-enriched class diagrams for three domains: Simulink, a project management system, and a phone system. These domains were identified from literature based on their familiarity to the authors (*convenience sampling* [252]). For the former two domains, we were aware of several available variants in the literature. To systematically identify available variants, we performed database searches in Google Scholar, IEEExplore, and ACM's Digital Library, with the search strings "Project Management meta-model" and "Simulink meta-model." The considered variants of *Phone* correspond to the feature model from the original paper.

Simulink [253] is a block-based modeling language that is widely applied in the design of embedded and cyber-physical systems. The absence of an official specification has given rise to the emergence of various variants. The *Project Management (PM)* [246] product line represents a family of software systems for project management, with concepts such as projects, activities, tasks, persons, and roles. The *Phone* product line, introduced by Benavides et al. [127], represents a family of software systems for mobiles phones with various hardware functionalities, such as different cameras and displays.

For each domain, we designed three class diagrams: one per variability mechanism. With three mechanisms (*Enu*, *Ann*, *com*) and three domains (*d1*, *d2*, *d3*), following a *Latin square design* [25, 26], the paths were:

- *Enu d1→Ann d2→Com d3* (path 1),
- *Com d1→Enu d2→Ann d3* (path 2), and

- *Ann d1→Com d2→Enu d3* (path 3).

We recruited 73 participants, all of which were BSc students in a German university. The students were recruited from courses with completed previous lectures and homework assignments on class models. In line with our strategy to recruit students familiar with class models, students expressed an average level of expertise, amounting to 3.47 (mean) ± 0.60 (standard deviation). The self-reported programming expertise of 3.62 ± 0.74 was comparable. In contrast, the self-reported expertise in variability mechanisms was considerably lower, amounting to 1.73 ±0.87 in annotative mechanisms, 1.86 ±1.03 in compositional mechanisms, and 1.87 ±0.93 in enumerative mechanisms. The homogenous experience of our participants is beneficial for the validity of our findings, by countering a possible threat related to different previous knowledge. In the light of our justification for selecting students as participants (based on evidence for their suitability as stand-ins for developers [245]), our sample is representative for developers with similar experience levels in modeling and variability mechanisms.

Participants were randomly divided into three groups, and assigned one of the above-mentioned paths, allowing them to experiment with each variability mechanism and each domain once. Each participant was required to perform three tasks types for the three models, each represented using a different variability mechanism. At the end, the participants were asked to provide their subjective assessments and preferences (S1–S6), and rationale for their choices.

Table 4.2: Correctness scores for experiment 1 (class diagrams).

	Annotative			Compositional			Enumerative		
Task type	**Mean**	**Mdn**	**Sdt.dev**	**Mean**	**Mdn**	**Std.ev**	**Mean**	**Mdn**	**St.dev**
1: Tracing elements to variants	1.7/2	2.0/2	0.6	1.5/2	2.0/2	0.7	1.7/2	2.0/2	0.6
2: Comparing two variants	1.5/2	1.5/2	0.6	1.5/2	1.5/2	0.6	1.6/2	1.5/2	0.4
3: Comparing all variants	1.6/2	2.0/2	0.7	1.2/2	1.0/2	0.7	1.5/2	2.0/2	0.7
Total	4.8/6	5.0/6	1.3	4.2/6	4.5/6	1.4	4.9/6	5.0/6	1.1

Mdn: Median, **St.dev:** Standard deviation

Table 4.3: Correctness scores for experiment 2 (state machine diagrams).

	Annotative			Compositional			Enumerative		
Task type	**Mean**	**Mdn**	**Sdt.dev**	**Mean**	**Mdn**	**Std.ev**	**Mean**	**Mdn**	**St.dev**
1: Tracing elements to variants	1.2/2	1.0/2	0.8	1.2/2	1.0/2	0.7	-	-	-
2: Comparing two variants	1.0/2	1.0/2	0.8	1.1/2	1.0/2	0.8	-	-	-
3: Comparing all variants	1.2/2	1.0/2	0.8	0.8/2	1.0/2	0.7	-	-	-
Total	3.4/6	3.0/6	2.0	3.1/6	3.0/6	1.8	-	-	-

Mdn: Median, **St.dev:** Standard deviation

Table 4.4: Correctness scores for experiment 3 (activity diagrams).

	Annotative			Compositional			Enumerative		
Task type	**Mean**	**Mdn**	**Sdt.dev**	**Mean**	**Mdn**	**St.dev**	**Mean**	**Mdn**	**St.dev**
1: Tracing elements to variants	1.4/2	1.5/2	0.6	1.2/2	1.0/2	0.6	-	-	-
2: Comparing two variants	1.4/2	1.5/2	0.6	0.9/2	1.0/2	0.7	-	-	-
3: Comparing all variants	1.4/2	1.5/2	0.5	0.9/2	1.0/2	0.6	-	-	-
Total	4.2/6	4.0/6	1.0	3.0/6	3.0/6	1.3	-	-	-

Mdn: Median, **St.dev:** Standard deviation

Experiment 2 considered state machine diagrams as model type. Both considered domains were based on the robotics programming game *Robocode*

[39]. The rationale was that students were involved with Robocode as a course project and were, therefore, familiar with it. Moreover, Robocode bots possessed considerable variability [155], which allowed us to model state machines that were reasonably complex. Our considered domains represented two high-level features of Robocode robots, each using a separate state machine diagram. For each domain, we designed two state machines: one using the annotative mechanism, and another one using the compositional mechanism.

We recruited 65 students from a Swedish university (63 Bsc, 2 PhD). Participants were divided into two groups, and assigned into those randomly. In the preliminary assessment, on average, the participants rated their experience with state machine diagrams to be 2.846 ±0.587. Participants also gave an average rating of 3.42 ±0.749 to their programming experience. The self-reported experience of participants for the two variability mechanisms was considerably lower: 2.26 ±0.73 for annotative, and 2.52 ±0.77 for compositional. These ratings are, however, comparable to each other, and do not indicate any bias towards a particular mechanism.

Following our *Latin square* design, we exposed both sub-domains (*d1*, *d2*) to the groups in the same order, and reversed the order of the variability mechanism they were represented with.

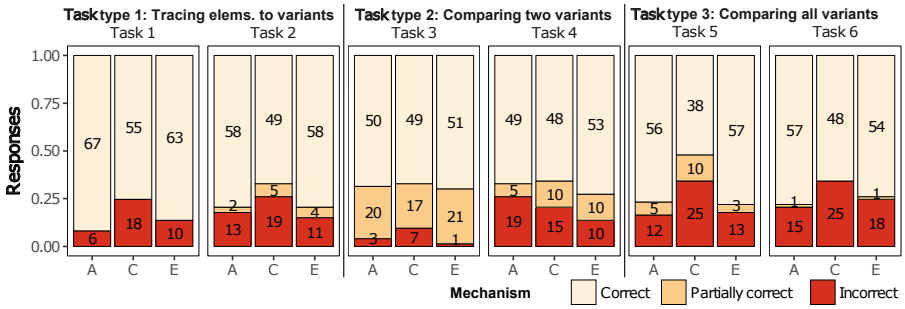
- *Com d1* →*Ann d2* (path 1),
- *Ann d1* →*Com d2* (path 2).

Experiment 3 was focused on *activity diagrams* as model type. We designed activity diagrams for *two* domains: a *Flight reservation system (FRS)* and an *Email service provider (ESP)*. We created them upon our experience and by taking inspiration from the literature.

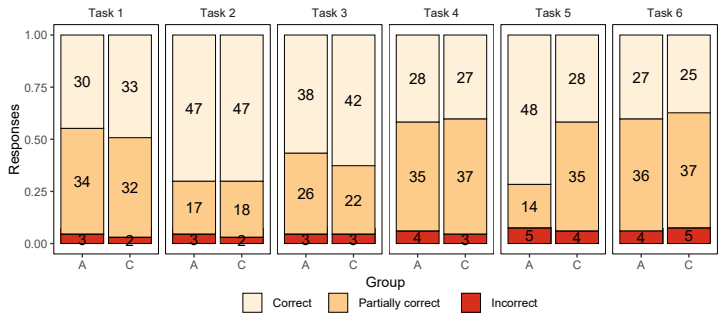
The participants in this experiment were *26 MSc students* studying at a Dutch university. Participants were randomly allocated into two groups. Participants indicated an average experience of 3.39 ±0.64 with activity diagrams, and 4.0 ±0.6 with programming. The mean scores for experiences with both variability mechanisms were significantly lower: 2.1 ±0.8 and 2.3 ±0.86 for annotative and compositional mechanisms respectively. The high ratings for experiences with activity diagrams and programming can be explained by our participants being MSc students, who have undergone significant practice with software design and implementation. The lower ratings for both variability mechanisms are also reasonable, since they were not taught in detail in the course. However, the ratings are close to one another, reducing any possible biases in the results.

We designed two activity diagrams per domain, each represented using a different variability mechanism. With two domains (*d1*, *d2*), the paths after applying our *Latin square* design were:

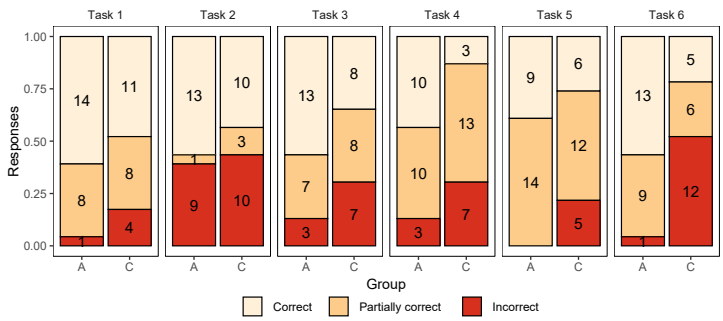
- *Ann d1*→*Com d2* (path 1).
- *Com d1*→*Ann d2* (path 2),



(a) Experiment 1



(b) Experiment 2



(c) Experiment 3

Figure 4.6: Correctness scores for all three experiments **A:** Annotative **C:** Compositional **E:** Enumerative)

4.6 Results

We now present the results from our experiments, structured along our research questions. For each, we have one subsection presenting the relevant results from all three experiments. This supports a direct comparison of our observations along our two independent variables (modeling language and variability mechanisms).

4.6.1 RQ1: Efficiency

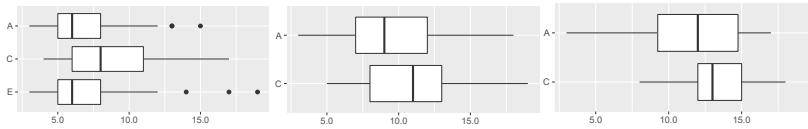
In RQ1, we studied the effect of annotative and compositional mechanisms on *efficiency*, that is, the ability of our participants to solve model comprehension tasks correctly and quickly. To this end, we computed correctness scores and completion time based on the task responses, collected in the *modules* of our questionnaire (see Figure 4.5). As explained in Section 4.5, the correctness score of a task type ranged between 0 and 2, based on aggregating the scores for the two questions in each task type. The completion time was determined by measuring the difference between the starting and ending time for completing all tasks for one particular variability mechanism.

Correctness. Table 4.2-4.4 provide a high-level overview of the correctness scores. Each table corresponds to one experiment and model type, and shows mean scores, median scores (Mdn), and standard deviations (St.dev) per task type and variability mechanism. A complementary, more detailed overview is offered by Figure 4.6a. Figure 4.6b, and Figure 4.6c, which visualize the distribution of scores obtained for each task.

Considering class diagrams (Experiment 1, Table 4.2), the participants generally performed equally well with the annotative and the enumerative mechanisms. In contrast, the use of the compositional mechanism lead to a noticeable drop in mean performance. Hypothesis testing showed that the difference for compositional to other types was significant for task types 1 and 3. For type 1, we found p=0.01 for the comparison to annotative, with a medium-ranged effect size of A_{12}=0.62 (p=0.02 for the comparison to enumerative, surpassing the corrected threshold). For type 3, we found p<0.01 when comparing compositional to both annotative and enumerative, with medium effect sizes (A_{12}=0.66 and 0.64, respectively). We did not find significant differences between the mechanisms for type 2. Annotative and enumerative do not differ significantly in any considered case.

The similar results obtained for annotative variability and the baseline (enumerative) for this and the following RQs motivated us to focus in experiments 2 and 3 on the comparison between annotative and compositional variability. Our rationale was that by not considering enumerative, we could allocate more time to annotative and compositional, which we could use for more involved questions within the task types that would allow a more in-depth comparison between these mechanisms.

Consequently, in Experiment 2 and Experiment 3, we observed lower average correctness scores than in Experiment 1. With state machine diagrams (Experiment 2, Table 4.3), there were no prominent differences between the results for annotative and compositional mechanisms for type 1 (1.2 vs 1.2) and 2 (1.0 vs 1.1). A significant difference was evident for the most complex task type, type 3,



(a) Completion times (Experiment 1) (b) Completion times (Experiment 2) (c) Completion times (Experiment 3)

Figure 4.7: Completion time in minutes for (a) Experiment 1 (b) Experiment 2 and c) Experiment 3. *Y-axis:* Variability mechanisms (**A:** Annotative **C:** Compositional **E:** Enumerative) *X-axis:* Time taken in minutes

where annotative outperformed compositional with a score of 1.2 vs. 0.8. For type 3, hypothesis testing revealed a significant difference in performance using annotative and compositional with p=0.01 and medium effect size (A_{12}=0.61). We did not find significant differences between the mechanisms for types 1 and 2. An explanation is offered by the nature of the considered examples: Answering the questions was possible without an in-depth understanding of how the composition operator works. Hence, an inherent disadvantage of the compositional mechanism does not manifest itself in these examples. For task type 3, where we find statistically different results, this concern does not apply.

With activity diagrams (Experiment 3, Table 4.4), the differences were more pronounced. Participants performed better using annotative mechanism for type 1 (1.4 vs 1.2), type 2 (1.4 vs 0.9) and type 3 (1.4 vs 0.9). Hypothesis testing reveals significant differences for all task types, with p<0.01 for all types. The effect size of this comparison was small-to-medium for type 1 (A_{12}=0.59), medium-to-large for type 2 (A_{12}=0.67) and large for type 3 (A_{12}=0.76). In the overall performance, the participants on average achieved a score 1.3 times higher with annotative than with compositional mechanism (4.2 vs 3.0). Compared to experiment 2, one noteworthy aspect that adds to the explanation of these findings is that performing the tasks required a better understanding of the involved composition algorithm, since relevant information was spread over several composition fragments.

Annotative outperformed compositional variability on average in all three experiments. Specifically, the annotative mechanism lead to higher correctness scores on average than the compositional one for all task types and domains except one. The differences were significant for six out of nine cases (3 task types and 3 experiments). Compared to the baseline (enumerative variability, Experiment 1), annotative variability did not lead to significant performance differences, whereas compositional variability did.

Completion Time. Table 4.5 provides an overview of the completion times of participants for solving all tasks in all experiments. According to this data, the average time taken for performing tasks using annotative mechanism was always less than time taken using other mechanisms. Figure 4.7a, Figure 4.7b, and Figure 4.7c represent the distribution of completeness times taken by participants using different variability mechanisms for Experiment 1, 2, and 3 respectively.

In line with the observations about correctness, the speed of performing tasks was highest in experiment 1, due to the simpler tasks in each task type, which al-

lowed us to consider the enumerative baseline solution in addition to annotative and compositional variability. The participants were fastest on average when using the annotative mechanism (mean completion time: 6.6 minutes), somewhat faster, but not significantly so than when using enumerative (7.1 minutes, p=0.12). Participants, however, tasks took significantly longer when using the compositional mechanism (8.8 minutes). The differences between compositional and both annotative and enumerative were highly significant with p<0.001. The effect size was large when comparing compositional and annotative (A_{12}=0.71) and medium-to-large for compositional to enumerative (A_{12}=0.67).

In experiment 2, participants took roughly the same amount of time on average using the annotative and compositional mechanisms (10.4 vs 11.3). Hypothesis testing showed that the difference between times taken using both mechanisms was not significant (p>0.25). These observations are in line with the correctness-related ones, in which only one of the considered task types (the most difficult one) lead to significant differences.

In experiment 3, participants were faster in performing their tasks on average using the annotative mechanism than with the compositional one (14.2 vs 16.8 minutes). However, this difference is not significant (p>0.38). Contrasted with the significant differences in the correctness scores for all three task types, we find that the tendencies of both observations agree, but the implications for correctness appear to be greater than those for completion time.

The annotative mechanism lead to the shortest on-average completion times in all experiments. Participants took longest to complete the tasks when using the compositional mechanism. Yet, only in one out of three experiments, statistical significance was found. Compared to the baseline (enumerative variability, Experiment 1), annotative variability outperformed the baseline solution, but not significantly so.

4.6.2 RQ2: Subjective Perception

We report on the participant's subjective perceptions of understandability and difficulty to complete the tasks, based on our subjective assessment questions (S1–S4 in our questionnaire; see Figure 4.5). The questions in this category were answered on a five-item Likert scale, with lower scores indicating better results. Table 4.6 gives an overview of the results, which are refined by the

Table 4.5: Completion times (in minutes) of our participants for all three experiments.

Exp	Mechanism	Min	Mn	Mdn	Max	St.dev
1	**Annotative**	3	6.6	6	15	2.6
	Compositional	4	8.8	8	17	3.2
	Enumerative	3	7.1	6	19	3.1
2	**Annotative**	3	10.4	10	23	4.3
	Compositional	5	11.3	11	22	3.7
3	**Annotative**	3	14.2	14	28	6.1
	Compositional	8	16.8	14	32	6.7

Mn: Mean, **Mdn:** Median, **St.dev:** Standard deviation

Table 4.6: Participant perceptions (understandability and difficulty ratings) for Experiment 1, 2, and 3.

			Annotative			Compositional			Enumerative		
Exp	**Quality**		**Mean**	**Mdn**	**St.dev**	**Mean**	**Mdn**	**St.dev**	**Mean**	**Mdn**	**St.dev**
1	**Understandability**		2.6/5	3/5	1.1	3.2/5	3/5	1.1	2.2/5	2/5	1.2
	Difficulty	Task type 1	2.3/5	2/5	1.2	3.1/5	3/5	1.2	2.3/5	2/5	1.1
		Task type 2	2.5/5	2/5	1.3	3.0/5	3/5	1.3	2.2/5	2/5	1.2
		Task type 3	2.5/5	2/5	1.2	3.2/5	3/5	1.3	2.5/5	2/5	1.1
2	**Understandability**		3.0/5	3/5	1.0	2.9/5	3/5	0.9	-	-	-
	Difficulty	Task type 1	2.6/5	3/5	1.0	2.6/5	3/5	1.0	-	-	-
		Task type 2	2.6/5	3/5	0.9	2.6/5	2/5	0.9	-	-	-
		Task type 3	2.9/5	3/5	1.0	2.9/5	3/5	1.0	-	-	-
3	**Understandability**		2.7/5	2/5	1.3	3.4/5	3/5	1.1	-	-	-
	Difficulty	Task type 1	2.3/5	2/5	1.0	3.2/5	3/5	0.9	-	-	-
		Task type 2	2.5/5	2/5	1.3	3.0/5	3/5	1.3	-	-	-
		Task type 3	2.5/5	2/5	1.2	3.2/5	3/5	1.3	-	-	-

[1] Scores on a 5-point Likert scale with 1: very easy, 5: very hard to understand.
[2] Scores on a 5-point Likert scale with 1: very easy, 5: very difficult to perform task.
Mdn: Median, **St.dev:** Standard deviation

visualizations in Figure 4.8a, 4.8b and 4.8c. The figures also include the exact formulations of all questions.

In Experiment 1 (class diagrams, Figure 4.8a), enumerative was considered to easiest to understand (mean: 2.2), followed by the annotative mechanism (mean: 2.6). Compositional mechanism was the hardest to understand (mean: 3.2). The differences between the *understandability* of all three mechanisms were significant, with varying effect size measures: for enumerative vs. annotative, p=0.006 with A_{12}=0.61 (small to medium effect); for compositional vs. annotative, p=0.004 with A_{12}=0.66 (medium effect); for enumerative vs. compositional, p$\leq$0.001, with A_{12}=0.73 (large effect).

The *difficulty* rating was fully consistent with both the objective task metrics (RQ1) and the understandability ratings. Regarding difficulty, annotative and enumerative were considered to be less difficult than compositional for all task types. Comparing the annotative and enumerative mechanism, the given mean ratings were approximately equal, amounting to 2.3, 2.5, 2.5 for annotative, and 2.3, 2.2, 2.5 for enumerative. We did not find statistical significance for this comparison, reinforcing our decision to not consider enumerative in the follow-up experiments. In contrast, the mean ratings for compositional of 3.1, 3.0 and 3.2 were much higher, indicating lower understandability. In all comparisons of compositional to another mechanism, we found significance. In all cases but one (task type 2, annotative vs. compositional: p=0.03; A_{12}=0.62), the p-value was below 0.003 and the effect size between A_{12}=0.65 and 0.69, indicating a medium-to-large effect.

In experiment 2 (state machine diagrams, Figure 4.8b), for understandability, participants gave approximately similar ratings to both annotative and compositional (3.0 vs. 2.9). The difference was not statistically significant. Considering difficulty, the mean scores per task type were identical (2.6, 2.6, 2.9 vs. 2.6, 2.6, 2.9), implying no statistically significant difference, largely consistent with the correctness scores and completion times for the same experiment (RQ1). Still, task type 3 gave rise to the only case in all our data where subjective assessment and objective performance were not aligned: the reported difficulty levels do not differ, while the correctness scores do so with statistical significance.

In experiment 3 (activity diagrams, Figure 4.8c), participants considered

annotative to be notably more understandable than compositional (2.7 vs. 3.4). The difference was statistically significant (p=0.3), with A_{12}=0.68 (medium-to-large effect size). Concerning difficulty, annotative was considered to be easier than compositional for all task types (2.3, 2.5, 2.5 vs 3.2, 3.0, 3.2). The difference for type 2 was significant, with p=0.005, and A_{12}=0.73 (large effect). These observations agreed with the correctness scores and completion times in RQ1.

The subjective perceptions of our participants largely agreed with the objective performance measurements: In two of the three experiments, the found the annotative mechanism more understandable and easier to work than the compositional one. The annotative mechanism was found (non-significantly) harder to understand, but equally easy to work with as the baseline (enumerative variability, Experiment 1).

Table 4.7: Distribution of preferred mechanisms per task type

Task type	Ann.	Com.	Enu.	None
Experiment 1				
1 Understanding variants	50.7%	13.7%	34.2%	1.4%
2 Comparing two variants	26.0%	15.1%	57.5%	1.4%
3 Comparing all variants	43.8%	12.3%	42.5%	1.4%
Experiment 2				
1 Understanding variants	58.6%	33.8%	-	7.6%
2 Comparing two variants	52.3%	41.5%	-	6.2%
3 Comparing all variants	46.2%	41.5%	-	12.3%
Experiment 3				
1 Understanding variants	78.3%	8.7%	-	13.0%
2 Comparing two variants	78.3%	21.7%	-	0%
3 Comparing all variants	78.3%	17.4%	-	4.3%

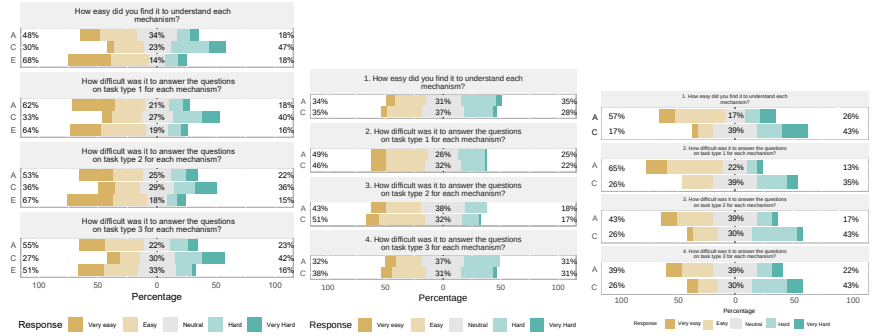
Ann: Annotative Variability **Com:** Compositional Variability **Enu:** Enumerative Variability

4.6.3 RQ3: Subjective Preferences

We report on our participants' preferences, based on questions S5 and S6 in our questionnaire (see Figure 4.5). In S5, we asked our participants to specify a preferred mechanism per task type (quantitative data). In S6, we asked them for textual feedback to explain the rationale for their preferences (qualitative data). In what follows, first, we present and discuss the distribution of preferences (S5), before explaining our observations based on the provided rationale (S6).

4.6.3.1 Quantitative Distribution of Subjective Preferences

Figure 4.9 and Table 4.7 provide an overview of our quantitative data: the percentages of selected answers when asked to specify a preferred variability mechanism per task type. For Experiment 1, we find that the preferences varied



(a) Experiment 1 (b) Experiment 2 (c) Experiment 3

Figure 4.8: Subjective perceptions of our participants. *Y-axis:* Variability mechanisms (**A:** Annotative **C:** Compositional **E:** Enumerative)

strongly between the tasks. Annotative was preferred by most participants for task types 1 and 3, albeit with only a moderate to slight difference to the enumerative mechanism: 50.7% vs. 34.2% for type 1, and 43.8% vs. 42.5% for type 3. In contrast, the enumerative mechanism was preferred with a large margin for task type 2, comparing two variants (which are explicitly present in the enumerative representation). Compositional came in last in all comparisons, with percentages between 12.3% and 15.1%. Participants generally expressed a preference; the *no-preference* option was only selected in 1.4% of all cases. Intuitively, the preference for enumerative for type 2 is not surprising: in a comparison between two variants, explicitly representing the variants seems beneficial. Based on the preference of annotative for type 1 and 3, we hypothesize that this representation is suitable for tasks that require a good overview of all variants and the ability to trace elements to variants.

For Experiment 2, participants preferences were consistent with their understandability ratings per mechanism. Participants preferred annotative over compositional for all task types: 58.6% vs. 33.8% for type 1, 52.3% vs. 41.5% for type 2, and 46.2% vs. 41.5% for type 3. The more pronounced preference of annotative for type 1 and 2 can be attributed to the consolidated view it offers, making it easier to understand and compare variants (also confirmed by the subjective assessments presented shortly). A considerable number of participants selected the *no-preference* option, especially for type 3 (12.3%). While this percentage is higher than in the other two experiments, one might still find it surprisingly low: Even though there were hardly any differences in the subjective assessment of understandability and difficulty (with low standard deviations), most participants still specified a preference. One could interpret this finding as supporting a role of personal *inclination* or *taste* in preferring a mechanism.

For Experiment 3, the differences were far more distinguished. The cleary majority of the participants preferred annotative over compositional for all task types (78.3% vs. 8.7% for type 1, 78.3% vs. 21.7% for type 2, and 78.3% vs. 17.4% for type 3). For type 1, 13% of the participants chose the *no-preference* option. The preferences were consistent with both the correctness scores and difficulty ratings. These observations are in line with our previous findings for this experiment, in which annotative was found easier to use and lead to better efficiency for all task types.

Compared to compositional, annotative was generally preferred for all task types, although with varying margins between the experiments. When also offered the baseline representation (enumerative, Experiment 1), the preferences varied between the task types: Annotative was preferred for comparing all variants, and for tracing elements to variants. Enumerative was largely preferred for comparing two variants to each other.

4.6.3.2 Qualitative Explanations of Subjective Preferences

To obtain additional insights, we asked the participants to explain their preferences intuitively using an open-ended question. Based on our manual assessment performed on the answers, we discuss the recurring aspects deemed as relevant by the participants below. Importantly, some of the reported aspects could be mitigated when working with proper tool support. Indeed, our findings might be useful for informing the question of what improved visualizations should focus on, as we further discuss in Section 4.6.4.

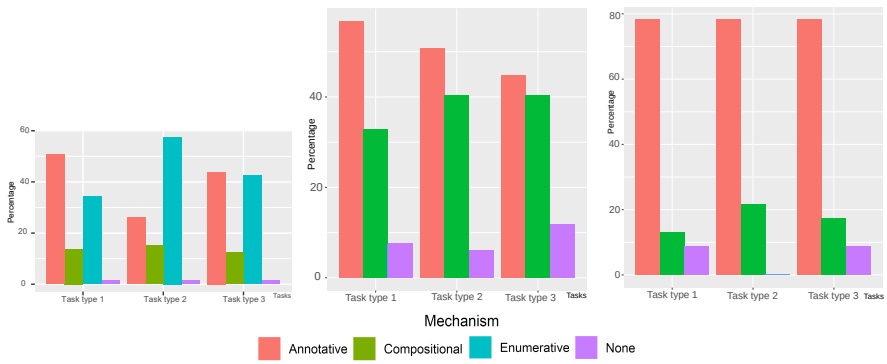


Figure 4.9: Preference distributions of our participants in Experiment 1–3 (from left to right)

Conciseness. Models created using the annotative mechanism offer a *consolidated* view, where all variants of the domain are concisely shown as a single model. Eight of our participants mentioned this as a benefit: *"In the annotative one you had all the information asked on the first look; comparing was easy since the different [variants] were all in the same diagram."* With compositional, the models were *modular* in nature. This however made the models *scattered*, and therefore, hard to deal with. Five of our participants remarked that the modularity made the models clearer and eased the tasks: *"It [was] easier to conduct direct comparisons between different products in the [compositional mechanism], due to its modular nature."* However, the scatteredness of the models was linked to difficulty in performing tasks: *"[using compositional, you] need to look at a lot of screens/windows."* A related aspect mentioned by four of our participants was that the *co-located* presentation of information made the information readily available, and helped in performing tasks: *"The enumerative shows all required parts at once and you don't have to look really close to see all required parts and connections."*

Understandability. Eight of our participants found enumerative mechanism easy to understand and work with: *"the enumerative variant is the easiest."* None of the participants found it difficult to deal with. 26 of our participants found annotative easier to understand and follow: *"Annotative is simple to understand and just beautiful."* Two participants remarked that annotative mechanism was intuitive: *"[Annotative] is ... way more intuitive, it gives a better overview."* In contrast, six participants found annotative to be hard to manage, mainly because of the *cluttered* models: *"The main reason why I had some challenges with the [annotative] version was probably because the text was so clustered and hard to read."* Five of our participants found compositional to be easier to understand, either because they were already familiar with the mechanism, or they liked the modular structure: *"It was easier to understand compositional because of their modularity in my opinion."* 15 of our participants experienced difficulty in understanding the compositional mechanism, mainly because of its scattered nature: *"The compositional one seems more scrambled, and therefore a bit harder to follow and get a complete picture of."*

Familiarity. We observed that familiarity is a factor in preferring one mechanism over the other, if only in a participant's initial assessment. Specifically, we observed two interesting aspects of familiarity. First, participants preferred mechanisms they had experience with: *"Since I haven't worked with [annotative] variability before it was easier to grasp the compositional version."* Second, once participants experienced with a particular mechanism, they changed their preferences along the way: *"At first the annotative was hard (it took a while to understand what the starting state was), but once I understood it, it was easier to handle than the compositional one."*

Scalability. In their explanation of why they preferred a particular variability mechanism, several participants extrapolated from the considered case to more complex ones, and foresaw potential issues with the scalability of the notation. *"[In enumerative,] although you need more models/space, you can see everything relatively easy. However, if you have maybe like 20 variants, enumerative is probably not the way to go."*, *"last the Enumerative, the 6 variants were okay but when there are even more it is to much" [sic]*, and finally *"The problem with the Compositional was [the] bigger and more complex it gets, it is harder to understand in a short time"*.

Efficiency. Participants preferred mechanisms which they found quicker to work with, based on their subjective impressions. We observed three relevant dimensions. First, participants favoured the mechanisms which offered *readiness*; having readily available information at one look made the tasks easier. The mechanisms favoured with this rationale were enumerative and annotative: *"The enumerative shows all required parts at once and you don't have to look really close to see all required parts and connections,"* and *"In [compositional], you have to follow a path to find the differences, where as with [annotative], you can see it right away."* Second, four participants expressed that the additional step of combining fragments to create variants in compositional was intensive: *"Since you do not have to think about how to compose diagrams I think that annotative is easier."* Third, the scattered nature of compositional models made the tasks repetitive, an aspect two of our participants found laborious: *"The tagged approach of [annotative] does not require me to mentally jump back and forth across the diagram like I have to do whilst using compositional diagrams."*

Labels. When creating models, we deliberately chose labels that differed from the names of the model elements to make the models reasonably complex. An example from the Email Service Provider domain is *Security*, which was the label used to map the activities *encrypt email* and *decrypt email*. Participants rendered labels to be useful, especially in annotative models: *"the labels with variant on classes [were] very [helpful]."* Labels in the models with annotative mechanism were even more favoured, owing to their arrangement, which is collocated with model elements: *"The feature placement in the annotative mechanism was more localized, making it feel better to work with."*

Colors. A design choice was to show the feature names in the annotative representation in distinguished colors, based on previous recommendations for code-level mechanisms [216]. Doing so balances out a disadvantage of annotative representations: the use of labels increases information density and visual crowding [254], thus affecting readability. In line with these findings, participants noted that it was *"easier to compare classes in Annotative because of colors"*, and that *"the colouring of annotative diagrams make [task type 1] really easy"*.

Overview. We observed that participants chose mechanisms which offered a good *overview* of the variants. Overview of variants is partially similar to our task type 1 (understanding variants), however, it has a broader relevance to other model-related tasks such as comparison. Six of our participants preferred annotative, explaining that it gave a good overview: *"It seems easier for me to ... get a fuller understanding of the system when reading the annotative mechanism."* Two participants preferred enumerative, and one participant compositional with this rationale. Two of our participants experienced difficulty in getting an overview with the compositional mechanism: *"There were many more places to look when comparing with the compositional mechanism and it became a bit difficult to overview."*

Flow. In both behavioural model types (state machine diagrams and activity diagrams), participants strongly preferred annotative, expressing that the *flow* of the model made the tasks easier. 11 participants expressed their preference for annotative: *"[It was] easier to understand how everything is connected with the annotative."* None of the participants found compositional to be helpful in understanding flow, seven expressing that compositional made it hard to perform tasks because it did not support understanding flow: *"I felt it was harder to navigate with the compositional statechart diagram."*

Task specific. Participants' preferences changed with task types. One participant commented: *"[task type 1 and task type 2] require more memorization about which features are active and which actions belong to them. As such, I prefer the compositional mechanism for these tasks, which is less cluttered and more cleanly displays only those sub-diagrams that are useful for a particular product. For the task type 3, it is likely that every feature will be enabled in at least one product. This means that you need to form a mental image of the diagram where all features are enabled. The fact that this is already contained in the annotative diagram is now an advantage. The compositional mechanism however, does not do this and as such requires more mental gymnastics in order to reason about a comparison of all products."*

Additional aspects. Two of our participants experienced *uncertainty*, both with compositional: *"At first glance Compositional seemed easy but I was never 100% sure about my answers concerning it."* Two of our participants expressed

that compositional mechanism assisted them to see only the *relevant* parts of the models: *"Compositional does the work of ignoring the unimportant parts for me; the boxes for each feature only contains the relevant state. ."* One participant observed that models are *harder to create* with the compositional mechanism: *"compositional based [...] is harder to construct and implement than the annotation based."* Two participants preferred annotative because it provided less information to be processed by the brain: *"[With compositional] you need to remember a lot of things, product names and it's features [..] Therefore I prefer the annotative notation, because you have to remember less mandatory features."* We propose related recommendations in Section 4.6.4.

A main contributing factor to the large preference of annotative was its conciseness; some inherent disadvantages could be balanced out by the use of labels and colors. Participant gave a variety of additional explanations, related to understandability, familiarity, scalability, efficiency, ability to provide a good overview, and understanding of flow.

4.6.4 Discussion and Recommendations

The objective and subjective differences between variability mechanisms observed in our study can be considered by tool and language developers for improving user experience, an important prerequisite for MDE adoption [255]. We discuss our findings in the light of derived recommendations.

Provide flexible, task-oriented representations. We find that there is no globally preferable variability mechanism—indeed, the "best" mechanism may depend on the task to be performed. Tool and language developers can support user performance and satisfaction by providing multiple representations, tailored to the task at hand. We propose to consider a spectrum of solutions, each trading off the desirable qualities *flexibility* and *simplicity*: As the most simple, but least flexible solution, one can augment a given representation with task-specific, read-only views, e.g., given an annotative representation, generate individual enumerated variants (or a subset thereof, see below). A second, more advanced solution is to make these additional representations editable, which offers more flexibility, but gives rise to a new instance of the well-known view-update problem [256]—-the particular challenge here is to deal with the implications of layout changes. The third, most advanced solution is *projectional editing* [257], in which developers interact with freely customizable representations of an underlying structure. Projectional editing offers the highest degree of flexibility, but poses a learning threshold to users for adapting to a new editing paradigm.

Support the simple solution, for appropriate use-cases. Our participants preferred the simple enumerative solution for a subset of tasks. While being commonly applied in practice (e.g., in 9 out of 23 cases studied by Tolvanen et al. [215]), this solution is inherently problematic: In small to moderate product lines, organizations struggle with the propagation of changes between cloned variants [202]. In large product lines, considering a distinct model for each of thousands of variants is simply infeasible. Instead, we suggest to address use-cases that involve a clearly defined subset of variants: In staged

configuration processes [258, 259], such subsets are derived by incrementally reducing the variant space, thus obtaining partial configurations of the system. Variability viewpoints [260], which are applied at companies like Daimler, reduce the variant space based on the perspective of a specific stakeholder. To address theses use cases, we suggest to provide support for selecting and interacting with a subset of enumerated variants, while using a proper variability mechanism for maintaining the overall system.

Use colors, and use them carefully. In line with the existing literature [216], we find that colors can be helpful for mitigating the drawbacks of annotative techniques. However, relying on colors in an unchecked way is undesirable due to the prevalence of color-blindness. Up to 8% of males and 0.5% of females of Northern European descent are affected by red-green color blindness [261]. A recommendation for language and tool designers is to avoid representations that solely rely on color, and to use dedicated color-blindness simulators such as Sim Daltonism (`https://michelf.ca/projects/sim-daltonism/`) to check their tools. A further issue with colors might be scalability, in a situation when there are many variation points that would need to be shown in different colors. A mitigation would be to flexibly reassign colors depending on the current screen focus, but even this approach would have limitations there is a need for a high number of annotation colors inside the same screen. Alternative visualization aids (for example, filtering) could be combined with coloring and are highly desirable.

Composition: the whole is more than the sum of its parts. Participants struggled with compositional mechanisms especially in tasks that required them to understand how fragments interact with each other. In the case of behavior modeling (Experiment 2 and 3), the observed performance differences are more pronounced when participants had to understand cross-fragment flow of activities or states. To be able to solve the tasks, they essentially had to execute the composition algorithm in their mind. Potentially, this drawback of compositional mechanisms can be balanced out by providing special visualizations that illustrate the composition of (subsets of) fragments. Such visualizations would reduce cognitive load during model comprehension and would also allow to provide instant feedback upon changes.

Structured overview of recommendations. Based on our results and the considerations in this section, we derive a structured overview of recommendations to tool developers and users, which we outline in Table 4.8. As the main factors that should determine the selection of the variability mechanism, we identify the *expected number of variants* and the *expected type of tasks*. This is because we saw different optimal (best-perfoming and preferred) mechanisms depending on the task type, and because the enumeration of variants, which works particularly well for comparing two variants, does not scale to large variant sets. To distinguish small from large variant sets, we use a size threshold of 10, based on the magnitude of the variant sets considered in our experiments. For small large sets, we propose to use either annotative or enumerative, depending on the tasks most likely to occur. For large sets, the considerations in the beginning of this section are particularly relevant: flexible, task-specific editing support can be used to show the information currently of interest. This helps to balance out scalability issues, notably the scalability issue with colors

in the annotative, and the issue with having an overly large set of variants in the enumerative mechanism.

Table 4.8: Overview of recommendations derived from our results.

Expected number of variants		**Expected type of tasks**		**Recommendation**
small (<10)	large (≥10)	comparing all variants, understanding individual variants	comparing two variants	
•		•		annotative or enumerative
•			•	enumerative
•		•	•	annotative or enumerative
	•	•		flexible
	•		•	flexible
	•	•	•	flexible

4.7 Threats to Validity

We discuss the threats to validity of our study, following the recommendations by Wohlin et al. [252].

External Validity. Our experiment focuses on class models, state machine diagrams, and activity diagrams, three ubiquitous model types. We discuss representativeness and practical relevance in MDE contexts in Sect. 2. Furthermore, we consider only two variability mechanisms in our comparison, an annotative and a compositional one. While studying a broader selection of modeling languages and mechanisms is left to future work, the qualitative data presented in Section 4.6.3.2 is not necessarily specific to these model types and mechanisms, and yields a promising outlook on generalizability.

Another issue is whether our results generalize to larger systems, specifically, those with more variants and model elements. Since the number of variants grows exponentially with the number of features, the enumerative representation will eventually be outperformed by the other ones. We discuss possible roles for the enumerative representation in larger systems in Section 4.6.4.

Finally, external validity is threatened by our sample of participants, made up of students with limited prior experience with variability mechanisms. Student participants can be representative stand-ins for practitioners in experiments that involve new development methods [245]. Still, the results could be different if considering participants that are experienced with the considered variability mechanisms. While considering a broader spectrum of experience levels would be worthwhile, we arguably focus on a critical population: In a given organization, consider the onboarding of a new team member with a similar experience level to our participants. Poor comprehension would pose a major hurdle to becoming productive, and, therefore, pose a risk for the organization. As mitigation measures, we established via our questionnaire that our participants had no unacceptable advantage of being much more familiar with either compositional or annotative variability, and selected subject domains that were either simple or already known to the students from previous course units (thus avoiding potential misunderstandings as a source of error).

Internal Validity. Within-subject designs help to elicit a representative number of data points to support statistically valid conclusions. We addressed their

drawbacks as follows: To address learning effects, we applied counterbalancing. Between the different groups, we distributed the order of variability mechanisms equally, while keeping the domain and task order constant. To balance the assignment of participants to classes, we randomized the assignment.

To avoid researcher bias in the selection of our examples, we selected domains that were not used before with a specific variability mechanism. Studying comprehension in larger models is desirable, but has some principle limitations with regard to the amount of information that participants can be exposed to in the scope of an experiment (participant fatigue).

A possible source of bias is our choice of subject domains: a particular domain might have been used in previous teaching units to explain a particular variability mechanism, which would give an unfair advantage to these mechanism. We intentionally selected domains that were simple and/or known to the students. However, to our knowledge, they were not used to teach particular variability mechanisms. Using our questionnaire, we asked our students to specify their previous knowledge about variability mechanisms, and did not find a noteworthy difference between the mechanisms.

Construct Validity. We operationalized comprehensibility with comprehension tasks, arguing for the importance of the considered tasks in Section 4.5. The choice of tasks was informed by our pre-study, in which we encountered trade-offs regarding participant fatigue and confounding factors when using more demanding tasks (Section 4.4). Since we find significant performance differences between mechanisms, the difficulty level of our tasks seems appropriate; however, other tasks might exist (e.g. understanding a single feature and its context), and task completion could also be facilitated if users are supported by specialized tools (e.g. query engines). Generally, systematic knowledge on the design space of model comprehension tasks would help to maximize realism in comprehension experiments, but such knowledge is currently lacking.

Our setup did not involve tools, representing an unavoidable trade-off: While having the participants use a tool environment would have been more realistic, it would have lead to confounding factors related to usability obstacles and participants' familiarity with the tool. On the other hand, working with a printout that could be derived from a tool (like in our experiments) also has some significant commonalities to working with the tool. We study the question of how well users can comprehend models in a particular visual representation. Given that the representation looks the same in the tool and on printout, we believe that construct validity for our results and recommendations is established. Notably, while adequate tool support might be able to mitigate some issues of particular representations, the developers of these tools should at least be aware of these issues.

Colors were only used in the annotative representation, where their usefulness (for distinguishing elements from different variants) seems more obvious than in the compositional one (where such elements are already distinguished by being contained in different modules). A follow-up study for studying the impact of colors in different representations might provide additional insight.

Subjective measures are generally less reliable than objective ones. However, previous findings suggest that they are correlated with objective performance measurements [262]. In fact, we find an agreement between the subjective and objective measurements performed in our experiments.

Conclusion Validity. Towards supporting conclusion validity, we used robust statistical tests. We report effect sizes and, to deal with threats related to multiple comparisons, applied a conservative correction to the considered significance threshold. Threats arising from random irrelevancies outside the experimental setting are reduced in experiments 1 and 2, which were administered with all participants in the same room. Experiment 3, administered during the COVID19 pandemic, is more prone to such irrelevancies, which, however, would equally apply to all treatments.

4.8 Related Work

Annotative vs. Compositional Variability. Annotative approaches are traditionally seen as inherently problematic. Spencer [219], for example, argues that #ifdef usage in C as a means to cope with variability is harmful, leading to convoluted, unreadable, and unmaintainable code (the infamous *"#ifdef hell"*). Spencer appeals to basic principles of good software engineering: explicit interfaces, information hiding, and encapsulation. Kästner et al. [218] argue that compositional approaches tend to promise advantages, which, however, only become manifest under rather specific assumptions. They emphasize that only empirical research can provide conclusive evidence.

Aleixo et al. [263] compare both mechanism types in the context of *Software Process Line* engineering, i.e., applying concepts and tools from SPL engineering to software processes. They compare two established tools: EPF Composer, which uses a compositional mechanism, and GenArch-P, which uses an annotative one. Similar to our conclusions, they report that the annotative mechanism performs better, especially with regards to a criterion the authors call *adoption*, i.e., how much knowledge is required to initially apply the mechanism.

Empirical Studies of Variability Mechanisms. Krüger et al. [17] present a comparative experimental study of two variability mechanisms: decomposition into classes, and annotations of code sections. They find that annotations have a positive effect on program comprehension, while the decomposition approach shows no significant improvement and, in some cases, a negative effect. While these findings are in line with ours, this study focuses on Java programs, and compares the considered mechanisms to a different baseline, pure OO code without any traces of variants. In our case, we considered the frequent case in industry of copied and reused model variants.

Fenske et al. [217] present an empirical study based on revision histories from eight open-source systems, in which they study the effect of #ifdef preprocessors to maintainability. They analyze maintainability in terms of change frequency, which is known to be correlated with error-proneness and change effort. In contrast to the traditional belief, they find that a negative effect of #ifdefs to maintainability cannot be confirmed.

Santos et al. [264] study the effect of using two code-level variability representations: feature-oriented programming (FOP) and conditional compilation (CC), on program comprehension. In their case, they focus on debugging task–a different scope than in our case. Specifically, they investigate the impact of FOP and CC on various maintenance tasks involving bug-finding. They conclude that there is no significant difference between the correctness, understanding,

and response time of using both representations.

Feigenspan et al. [216] study the potential of background colors as an aid to support program comprehension of source code with #ifdef preprocessors. In three controlled experiments with varying tasks and program sizes, they find that background colors contributed to better program comprehension and were preferred by the participants. We base the use of color in our experiments on these findings, and confirm them for the previously unconsidered case of a model-level variability mechanism.

Empirical Studies of Model Comprehension. Labunets et al. [33] study graphical and tabular models representations in security risk assessment. In two experiments, they find that participants prefer both representations to a similar degree, but perform significantly better when using the tabular one. The authors build on cognitive fit theory [227] to explain their findings: tables represent the data in a more suitable way for the considered task. Like we do, this study supports the need for task-tailored representations.

Nugroho [35] studies the effect of *level of detail* (LoD) on model comprehension. In an experimental evaluation with students, the author finds that a more detailed representation contributes to improved model understanding. Ramadan et al. [228] find a positive effect to comprehension of security and privacy aspects when graphical annotations are included in the considered models. Our results are in line with these findings, since the annotative mechanism includes the names of the associated variants as one point of additional information.

Acreţoaie et al. [34] empirically assess three model transformation languages with regard to comprehensibility. They consider a textual language and two graphical ones, one of which uses stereotype annotations to specify change actions in UML diagrams. They observe best completion times and lowest cognitive load when using the graphical language with annotations, and best correctness when using the textual language. Studying this trade-off further, by studying variability mechanisms in graphical and textual representations, would be an interesting extension of our work.

Model-level variability. There are two main research directions of variability engineering at the model level: variability modeling and model-level variability mechanisms. The former focuses on the modeling of the problem space (e.g., features and their relationships); the latter, which is the scope of this paper, focuses on the solution space (implementation of variability in domain models). Regarding variability modeling, there is a number of experience reports from various domains. Alférez et al. [21] observe that existing variability modeling approaches do not suffice to capture a number of aspects of video domain, including numeric parameters, multifeatures and constraints. They qualitatively compare a set of 13 approaches based on their ability to support the above-mentioned aspects, and present a new textual variability modeling language (VM) for modeling videos. Berger et al. investigate variability modeling of topological spaces [143]. They share their experiences from modeling large-scale fire alarm systems using UML2 class diagrams, and show that class diagrams are an effective tool to model topological variability to generate configurator tools. García et al. [142] give an experience report on the modeling of variability in robotic applications. They use feature models as a notation for defining variability of robotic applications in various dimensions, such as environment, hardware, and mission variability.

Most existing work on model-level variability mechanisms focuses either on particular mechanisms (see Section 4.2) or the use of such mechanisms for other tasks; e.g., product line testing [244] and product derivation [265]. Kühn et al. [230] compare two particular approaches to variability engineering in DSLs along three dimensions–feasibility, scalability, and sustainability—based on intuitive arguments. In summary, the impact of variability mechanisms on model comprehension has not been investigated yet in an empirical study. Our study is the first to use a controlled experiment to produce systematic evidence on the impact of variability representations on model comprehension, and as such, can be used to guide the choice of variability mechanisms when modelling the static and dynamic structure of product lines.

4.9 Conclusion

We presented a family of controlled experiments, in which we studied the effect of the variability mechanisms of two fundamental kinds—annotative and compositional mechanisms—on model comprehension. Our considered model types—class diagrams, state machine diagrams, and activity diagrams—are among the most popular and common models used in software engineering. We conducted the study with student participants with relevant background knowledge. For models with a scope and size similar to our examples and for similar tasks, we can conclude that:

- Annotative variability resulted in better comprehensibility than compositional variability for all task types.
- Compositional mechanism can impair comprehensibility in tasks that require a good overview of all variants.
- Annotative variability is preferred over the compositional one by a majority of the participants for all task types in all model types.
- The preferred variability mechanism depends on the task at hand.

We presented several recommendations to language and tool developers. We discuss a spectrum of solutions for maintaining several task-tailored representations. Having such solutions is especially important in large systems where maintaining a separate model per variant is infeasible, but developers might still want to interact with (sets of) variants of interest in a particular task. We endorse the recommendation to use colors for improving comprehension in annotative variability, and discuss its limitations. If a compositional mechanism is desired, users should be supported with visualizations, instead of being required to perform composition in their minds.

We envision four directions of future work. First, we want to understand the effect of tools to model comprehension. Second, we wish to systematically explore the space of typical tasks during model comprehension. Additional experiments would allow us to come up with a catalog of task-specific recommendations for variability mechanism use. Third, we are interested in broadening the scope of our experiments to take different modeling languages into account, including textual ones, which represent a middle ground between

traditional programming languages and graphical modeling languages, and transformation languages, for which many different reuse mechanisms have recently been developed [114, 266]. Fourth, further insight from our collected data could be obtained by performing sub-group analysis. Questions of interest are the effect of prolonged interaction of developers with different model types on their performance for the three task types, and the effect of mechanisms on the types of errors made by our participants (e.g., false negatives vs. false positives).

Acknowledgement. We thank the reviewers and the participants from our pre- and three main experiments, as well as the reviewers for their valuable comments. This work is supported by the Swedish Research Council and the Wallenberg Academy.

Bibliography

[1] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, t. Stănciulescu, A. Wąsowski, and I. Schaefer, "Flexible product line engineering with a virtual platform," in *Companion Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 532–535.

[2] F. J. Van der Linden, K. Schmid, and E. Rommes, *Software product lines in action: the best industrial practice in product line engineering.* Springer Science & Business Media, 2007.

[3] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An exploratory study of cloning in industrial software product lines," in *2013 17th European Conference on Software Maintenance and Reengineering.* IEEE, 2013, pp. 25–34.

[4] H. P. Jepsen, J. G. Dall, and D. Beuche, "Minimally invasive migration to software product lines," in *11th International Software Product Line Conference (SPLC 2007).* IEEE, 2007, pp. 203–211.

[5] J. Rubin, K. Czarnecki, and M. Chechik, "Managing cloned variants: a framework and experience," in *Proceedings of the 17th International Software Product Line Conference*, 2013, pp. 101–110.

[6] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Enhancing clone-and-own with systematic reuse for developing software variants," in *2014 IEEE International Conference on Software Maintenance and Evolution.* IEEE, 2014, pp. 391–400.

[7] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Bottom-up technologies for reuse: automated extractive adoption of software product lines," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C).* IEEE, 2017, pp. 67–70.

[8] T. Pfofe, T. Thüm, S. Schulze, W. Fenske, and I. Schaefer, "Synchronizing software variants with variantsync," in *Proceedings of the 20th International Systems and Software Product Line Conference*, 2016, pp. 329–332.

[9] L. Montalvillo and O. Díaz, "Tuning github for spl development: branching models & repository operations for product engineers," in *Proceedings of the 19th International Conference on Software Product Line*, 2015, pp. 111–120.

[10] 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. 17, no. 5, pp. 627–646, 2015.

[11] L. M. Northrop, "Sei's software product line tenets," *IEEE software*, vol. 19, no. 4, pp. 32–40, 2002.

[12] K. Pohl, G. Böckle, and F. Van Der Linden, *Software product line engineering: foundations, principles, and techniques.* Springer, 2005, vol. 1.

[13] K. C. Kang, J. Lee, and P. Donohoe, "Feature-oriented product line engineering," *IEEE software*, vol. 19, no. 4, pp. 58–65, 2002.

[14] K. Czarnecki, "Overview of generative software development," in *International Workshop on Unconventional Programming Paradigms.* Springer, 2004, pp. 326–341.

[15] S. Apel, D. Batory, C. Kästner, and G. Saake, "Software product lines," in *Feature-Oriented Software Product Lines.* Springer, 2013, pp. 3–15.

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

[17] J. Krüger, G. Calikli, T. Berger, T. Leich, and G. Saake, "Effects of explicit feature traceability on program comprehension," 2019.

[18] S. Apel, F. Janda, S. Trujillo, and C. Kästner, "Model superimposition in software product lines," in *ICMT.* Springer, 2009, pp. 4–19.

[19] F. Heidenreich, J. Kopcsek, and C. Wende, "Featuremapper: mapping features to models," in *ICSE-Companion.* ACM, 2008, pp. 943–944.

[20] F. Schwägerl, T. Buchmann, and B. Westfechtel, "Supermod—a model-driven tool that combines version control and software product line engineering," in *ICSOFT*, vol. 2. IEEE, 2015, pp. 1–14.

[21] M. Alférez, M. Acher, J. A. Galindo, B. Baudry, and D. Benavides, "Modeling variability in the video domain: language and experience report," *Software Quality Journal*, vol. 27, no. 1, pp. 307–347, 2019.

[22] A. Anjorin, K. Saller, M. Lochau, and A. Schürr, "Modularizing triple graph grammars using rule refinement," in *FASE*, 2014, pp. 340–354.

[23] A. Mehmood and D. N. Jawawi, "Aspect-oriented model-driven code generation: A systematic mapping study," *Information and Software Technology*, vol. 55, no. 2, pp. 395–411, 2013.

[24] T. Ahmad, J. Iqbal, A. Ashraf, D. Truscan, and I. Porres, "Model-based testing using uml activity diagrams: A systematic mapping study," *Computer Science Review*, vol. 33, pp. 98–112, 2019.

[25] J. Melo, C. Brabrand, and A. Wąsowski, "How does the degree of variability affect bug finding?" in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 679–690.

[26] D. C. Montgomery, *Design and analysis of experiments*. John wiley & sons, 2017.

[27] 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*, 2018.

[28] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Clone management for evolving software," *IEEE transactions on software engineering*, vol. 38, no. 5, pp. 1008–1026, 2011.

[29] J. E. Mathieu, M. A. Marks, and S. J. Zaccaro, "Multiteam systems." 2002.

[30] B. Tan Erciyes, S.-Y. Lim, S. Um, and E. Anderson, "Do frequent platform versions benefit platform developers and owners?" 2020.

[31] C. J. R. RÖßGER, G. ROCK, K. THEIS, C. WEIDENBACH, and P. WISCHNEWSKI, "Model-based variant management with v. control," in *Transdisciplinary Lifecycle Analysis of Systems: Proceedings of the 22nd ISPE Inc. International Conference on Concurrent Engineering, July 20-23, 2015*, vol. 2. IOS Press, 2015, p. 194.

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

[33] K. Labunets, F. Massacci, F. Paci, S. Marczak, and F. M. de Oliveira, "Model comprehension for security risk assessment: an empirical comparison of tabular vs. graphical representations," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3017–3056, 2017.

[34] V. Acreţoaie, H. Störrle, and D. Strüber, "Vmtl: a language for end-user model transformation," *Software & Systems Modeling*, vol. 17, no. 4, pp. 1139–1167, 2018.

[35] A. Nugroho, "Level of detail in uml models and its impact on model comprehension: A controlled experiment," *Information and Software Technology*, vol. 51, no. 12, pp. 1670–1685, 2009.

[36] S. Zschaler, P. Sánchez, J. Santos, M. Alférez, A. Rashid, L. Fuentes, A. Moreira, J. Araújo, and U. Kulesza, "Vml*–a family of languages for variability management in software product lines," in *International Conference on Software Language Engineering*. Springer, 2009, pp. 82–102.

[37] 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.

[38] B. A. Kitchenham, D. Budgen, and P. Brereton, *Evidence-based software engineering and systematic reviews*. CRC press, 2015, vol. 4.

[39] K. Hartness, "Robocode: using games to teach artificial intelligence," *Journal of Computing Sciences in Colleges*, vol. 19, no. 4, pp. 287–291, 2004.

[40] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich, "Featureide: An extensible framework for feature-oriented software development," *Science of Computer Programming*, vol. 79, pp. 70–85, 2014.

[41] 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.

[42] J. Martinson, H. Jansson, M. Mukelabai, T. Berger, A. Bergel, and T. Ho-Quang, "Hans: Ide-based editing support for embedded feature annotations," in *Proceedings of the 25th ACM International Systems and Software Product Line Conference-Volume B*, 2021, pp. 28–31.

[43] 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.

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

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

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

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

[48] 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.

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

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

[51] 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.

[52] 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.

[53] D. Nešić, J. Krüger, t. Stănciulescu, and T. Berger, "Principles of feature modeling," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 62–73.

[54] T. Berger, S. She, R. Lotufo, K. Czarnecki, and A. Wasowski, "Feature-to-code mapping in two large product lines." in *SPLC*. Citeseer, 2010, pp. 498–499.

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

[56] 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.

[57] 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 *Proceedings of the 2nd International Workshop on Product Line Approaches in Software Engineering*, 2011, pp. 20–24.

[58] W. K. 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.

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

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

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

[62] 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.

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

[64] 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.

[65] 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.

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

[67] 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.

[68] 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.

[69] 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.

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

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

[72] B. Zhang, M. Becker, T. Patzke, K. Sierszecki, and J. E. Savolainen, "Variability evolution and erosion in industrial product lines: a case study," in *Proceedings of the 17th International Software Product Line Conference*, 2013, pp. 168–177.

[73] 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.

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

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

[76] 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.

[77] 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 *Proceedings of the 14th International Conference on Modularity*, 2015, pp. 81–92.

[78] 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 *Proceedings of the 19th International Conference on Software Product Line*, 2015, pp. 16–25.

[79] J. Krueger, 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 *Twelfth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, 2018.

[80] 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 & Software*, vol. 152, pp. 239–253, 2019.

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

[82] 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.

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

[84] 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.

[85] 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.

[86] 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.

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

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

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

[90] 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.

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

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

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

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

[95] 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.

[96] 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.

[97] 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.

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

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

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

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

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

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

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

[105] 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.

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

[107] 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.

[108] D. Wille, T. Runge, C. Seidl, and S. Schulze, "Extractive software product line engineering using model-based delta module generation," in *VaMoS*. ACM, 2017, pp. 36–43.

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

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

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

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

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

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

[115] 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.

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

[117] B. Andam, A. Burger, T. Berger, and M. R. Chaudron, "Florida: Feature location dashboard for extracting and visualizing feature traces," in *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems*, 2017, pp. 100–107.

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

[119] H. Abukwaik, A. Burger, B. K. Andam, and T. Berger, "Semi-automated feature traceability with embedded annotations," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 529–533.

[120] M. Lillack, Ştefan Stănciulescu, W. Hedman, T. Berger, and A. Wąsowski, "Intention-based integration of software variants," in *ICSE*, 2019.

[121] C. Gacek and M. Anastasopoules, "Implementing product line variabilities," in *SSR*. ACM, 2001.

[122] J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, T. Leich, and G. Saake, *Mastering Software Variability with FeatureIDE*. Springer, 2017.

[123] D. Beuche, "Variants and variability management with pure::variants," in *SPLC*, 2004.

[124] C. W. Krueger and P. C. Clements, "Systems and software product line engineering with biglever software gears," in *SPLC*. ACM, 2013.

[125] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wąsowski, "Cool features and tough decisions: A comparison of variability modeling approaches," in *VaMoS*. ACM, 2012, pp. 173–182.

[126] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux, "Feature diagrams: A survey and a formal semantics," in *RE*. IEEE, 2006.

[127] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Information systems*, vol. 35, no. 6, pp. 615–636, 2010.

[128] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A Classification and Survey of Analysis Strategies for Software Product Lines," *ACM Computing Surveys*, vol. 47, no. 1, 2014.

[129] M. Mukelabai, D. Nesic, S. Maro, T. Berger, and J.-P. Steghöfer, "Tackling combinatorial explosion: A study of industrial needs and practices for analyzing highly configurable systems," in *ASE*. ACM, 2018.

[130] M. Meyer, "Continuous integration and its tools," *IEEE Software*, vol. 31, no. 3, 2014.

[131] D. Ståhl, K. Hallén, and J. Bosch, "Continuous integration and delivery traceability in industry: Needs and practices," in *SEAA*. IEEE, 2016.

[132] C. Krueger, "Easing the transition to software mass customization," in *International Workshop on Software Product-Family Engineering*. Springer, 2001, pp. 282–293.

[133] P. Ralph, "Toward methodological guidelines for process theories and taxonomies in software engineering," *Transactions on Software Engineering*, vol. 45, no. 7, 2019.

[134] D. I. K. Sjøberg, T. Dybå, B. C. D. Anda, and J. E. Hannay, "Building theories in software engineering," in *Guide to Advanced Empirical Software Engineering*. Springer, 2008.

[135] B. Fitzgerald and K.-J. Stol, "Continuous software engineering and beyond: Trends and challenges," in *RCoSE*. ACM, 2014.

[136] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wasowski, "Three cases of feature-based variability modeling in industry," in *International Conference Model-Driven Engineering Languages and Systems (MODELS)*, 2014, pp. 302–319.

[137] R. Rabiser, K. Schmid, M. Becker, G. Botterweck, M. Galster, I. Groher, and D. Weyns, "A study and comparison of industrial vs. academic software product line research published at splc," in *SPLC*, 2018.

[138] H. P. Jepsen and D. Beuche, "Running a software product line: Standing still is going backwards," in *SPLC*. ACM, 2009.

[139] H. P. Jepsen and F. Nielsen, "A two-part architectural model as basis for frequency converter product families," in *IW-SAPF*. Springer, 2000.

[140] J. Krüger, T. Berger, and T. Leich, *Software Engineering for Variability Intensive Systems*. CRC Press, 2019, ch. Features and How to Find Them: A Survey of Manual Feature Location.

[141] C. Seidl, T. Berger, C. Elsner, and K.-B. Schultis, "Challenges and solutions for opening small and medium-scale industrial sofware platforms," in *SPLC*. ACM, 2017.

[142] S. Garcia, D. Strueber, D. Brugali, A. D. Fava, P. Schillinger, P. Pelliccione, and T. Berger, "Variability modeling of service robots: Experiences and challenges," in *VaMoS*. ACM, 2019.

[143] T. Berger, S. Stănciulescu, O. Ogaard, O. Haugen, B. Larsen, and A. Wąsowski, "To connect or not to connect: Experiences from modeling topological variability," in *SPLC*. IEEE, 2014.

[144] R. Lindohf, J. Krüger, E. Herzog, and T. Berger, "Software product-line evaluation in the large," *Empirical Software Engineering*, 2020.

[145] E. Kuiter, J. Krüger, S. Krieter, T. Leich, and G. Saake, "Getting rid of clone-and-own: Moving to a software product line for temperature monitoring," in *SPLC*. ACM, 2018.

[146] K. Schmid and M. Verlage, "The Economic Impact of Product Line Adoption and Evolution," *IEEE Software*, vol. 19, no. 4, 2002.

[147] P. C. Clements and C. W. Krueger, "Point/Counterpoint: Being Proactive Pays Off / Eliminating the Adoption Barrier," *IEEE Software*, vol. 19, no. 4, 2002.

[148] J. Krüger, W. Fenske, J. Meinicke, T. Leich, and G. Saake, "Extracting software product lines: A cost estimation perspective," in *SPLC*. ACM, 2016.

[149] J. Krüger, "A cost estimation model for the extractive software-product-line approach," Master's thesis, University of Magdeburg, 2016.

[150] B. Curtis, M. I. Kellner, and J. Over, "Process modeling," *Communications of the ACM*, vol. 35, no. 9, 1992.

[151] H. Snyder, "Literature review as a research methodology: An overview and guidelines," *Journal of Business Research*, vol. 104, 2019.

[152] J. Krüger, C. Lausberger, I. von Nostitz-Wallwitz, G. Saake, and T. Leich, "Search. review. repeat? an empirical study of threats to replicating slr searches," *Empirical Software Engineering*, vol. 25, no. 1, 2020.

[153] Y. Shakeel, J. Krüger, I. von Nostitz-Wallwitz, C. Lausberger, G. Campero Durand, G. Saake, and T. Leich, "(automated) literature analysis - threats and experiences," in *SE4Science*. ACM, 2018.

[154] S. Imtiaz, M. Bano, N. Ikram, and M. Niazi, "A tertiary study: Experiences of conducting systematic literature reviews in software engineering," in *EASE*. ACM, 2013.

[155] J. Martinez, X. Tërnava, and T. Ziadi, "Software product line extraction from variability-rich systems: The robocode case study," in *SPLC*. ACM, 2018.

[156] T. Yue, S. Ali, and B. Selic, "Cyber-physical system product line engineering: Comprehensive domain analysis and experience report," in *SPLC*. ACM, 2015.

[157] J. H. Weber, A. Katahoire, and M. Price, "Uncovering variability models for software ecosystems from multi-repository structures," in *VaMoS*. ACM, 2015.

[158] H. Koziolek, T. Goldschmidt, T. de Gooijer, D. Domis, S. Sehestedt, T. Gamer, and M. Aleksy, "Assessing software product line potential: An exploratory industrial case study," *Empirical Software Engineering*, vol. 21, no. 2, 2016.

[159] M. Nagamine, T. Nakajima, and N. Kuno, "A case study of applying software product line engineering to the air conditioner domain," in *SPLC*. ACM, 2016.

[160] T. Iida, M. Matsubara, K. Yoshimura, H. Kojima, and K. Nishino, "Ple for automotive braking system with management of impacts from equipment interactions," in *SPLC*. ACM, 2016.

[161] R. Capilla and J. Bosch, "Dynamic variability management supporting operational modes of a power plant product line," in *VaMoS*. ACM, 2016.

[162] M. Usman, M. Z. Iqbal, and M. U. Khan, "A product-line model-driven engineering approach for generating feature-based mobile applications," *Journal of Systems and Software*, vol. 123, 2017.

[163] S. P. Gregg, D. M. Albert, and P. C. Clements, "Product line engineering on the right side of the "V"," in *SPLC*. ACM, 2017.

[164] B. Young, J. Cheatwood, T. Peterson, R. Flores, and P. Clements, "Product line engineering meets model based engineering in the defense and automotive industries," in *SPLC*. ACM, 2017.

[165] K. Hayashi, M. Aoyama, and K. Kobata, "Agile tames product line variability: An agile development method for multiple product lines of automotive software systems," in *SPLC*. ACM, 2017.

[166] A. Cortiñas, M. R. Luaces, O. Pedreira, Á. S. Places, and J. Pérez, "Web-based geographic information systems sple: Domain analysis and experience report," in *SPLC*. ACM, 2017.

[167] R. Pohl, M. Höchsmann, P. Wohlgemuth, and C. Tischer, "Variant management solution for large scale software product lines," in *ICSE-SEIP*. ACM, 2018.

[168] K. Hayashi and M. Aoyama, "A multiple product line development method based on variability structure analysis," in *SPLC*. ACM, 2018.

[169] M. Sayagh, N. Kerzazi, B. Adams, and F. Petrillo, "Software configuration engineering in practice: Interviews, survey, and systematic literature review," *IEEE Transactions on Software Engineering*, 2018.

[170] L. Marchezan, E. Macedo Rodrigues, M. Bernardino, and F. Paulo Basso, "Paxspl: A feature retrieval process for software product line reengineering," *Software: Practice and Experience*, vol. 49, no. 8, 2019.

[171] J.-M. Horcas, M. Pinto, and L. Fuentes, "Software product line engineering: A practical experience," in *SPLC*. ACM, 2019.

[172] L. Linsbauer, S. Malakuti, A. Sadovykh, and F. Schwägerl, "1st intl. workshop on variability and evolution of software-intensive systems (varivolution)," in *SPLC*, 2018.

[173] M. Nieke, L. Linsbauer, J. Krüger, and T. Leich, "Second international workshop on variability and evolution of software-intensive systems (varivolution 2019)," in *SPLC*, 2019.

[174] J. Krüger, S. Ananieva, L. Gerling, and E. Walkingshaw, "Third international workshop on variability and evolution of software-intensive systems (varivolution 2020)," in *SPLC*. ACM, 2020.

[175] Object Management Group, *Unified Modeling Language: Superstructure Version 2.1.1*, 2007.

[176] M. S. Ali, M. A. Babar, and K. Schmid, "A comparative survey of economic models for software product lines," in *SEAA*. IEEE, 2009.

[177] A. J. Nolan and S. Abrahão, "Dealing with cost estimation in software product lines: Experiences and future directions," in *SPLC*. Springer, 2010.

[178] P. M. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*. Pearson, 2007.

[179] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, "Devops," *IEEE Software*, vol. 33, no. 3, 2016.

[180] J. Krüger, "Are you talking about software product lines? an analysis of developer communities," in *VaMoS*. ACM, 2019.

[181] M. Staples and D. Hill, "Experiences adopting software product line development without a product line architecture," in *APSEC*. IEEE, 2004.

[182] J. Rubin, A. Kirshin, G. Botterweck, and M. Chechik, "Managing forked product variants," in *SPLC*. ACM, 2012.

[183] J. Rubin and M. Chechik, "A framework for managing cloned product variants," in *ICSE*. IEEE, 2013.

[184] J. Thönes, "Microservices," *IEEE Software*, vol. 32, no. 1, 2015.

[185] W. Wolf, "Cyber-physical systems," *Computer*, no. 3, 2009.

[186] Y. Lu, "Industry 4.0: A survey on technologies, applications and open research issues," *Journal of Industrial Information Integration*, vol. 6, 2017.

[187] T. Dillon, C. Wu, and E. Chang, "Cloud computing: Issues and challenges," in *AINA*. IEEE, 2010.

[188] W. K. G. Assunção, J. Krüger, and W. D. F. Mendonça, "Variability management meets microservices: Six challenges of re-engineering microservice-based webshops," in *SPLC*. ACM, 2020.

[189] J. Krüger, S. Nielebock, S. Krieter, C. Diedrich, T. Leich, G. Saake, S. Zug, and F. Ortmeier, "Beyond software product lines: Variability modeling in cyber-physical systems," in *SPLC*. ACM, 2017.

[190] A. Moran, *Managing Agile*. Springer, 2015.

[191] B. Meyer, *Agile!* Springer, 2014.

[192] E. S. de Almeida, A. Alvaro, D. Lucrédio, V. C. Garcia, and S. R. de Lemos Meira, "A survey on software reuse processes," in *IRI*. IEEE, 2005.

[193] W. B. Frakes and K. Kang, "Software reuse research: Status and future," *IEEE Transactions on Software Engineering*, vol. 31, no. 7, 2005.

[194] C. W. Krueger, "Software reuse," *ACM Computing Surveys*, vol. 24, no. 2, 1992.

[195] V. Bauer and A. Vetro', "Comparing reuse practices in two large software-producing companies," *Journal of Systems and Software*, vol. 117, 2016.

[196] M. C and K. Chandrasekaran, "Systematic studies in software product lines: A tertiary study," in *SPLC*. ACM, 2017.

[197] W. Fenske, T. Thüm, and G. Saake, "A taxonomy of software product line reengineering," in *VaMoS*. ACM, 2013, pp. 4:1–8.

[198] T. Berger, R.-H. Pfeiffer, R. Tartler, S. Dienst, K. Czarnecki, A. Wasowski, and S. She, "Variability mechanisms in software ecosystems," *Information and Software Technology*, vol. 56, no. 11, pp. 1520–1535, 2014.

[199] J. Van Gurp, J. Bosch, and M. Svahnberg, "On the notion of variability in software product lines," in *ICSA*. IEEE, 2001, pp. 45–54.

[200] M. Famelis, L. Lúcio, G. Selim, A. Di Sandro, R. Salay, M. Chechik, J. R. Cordy, J. Dingel, H. Vangheluwe, and S. Ramesh, "Migrating automotive product lines: a case study," in *International Conference on Theory and Practice of Model Transformations*. Springer, 2015, pp. 82–97.

[201] R. Flores, C. Krueger, and P. Clements, "Mega-Scale Product Line Engineering at General Motors," in *Proc. SPLC*, 2012.

[202] A. Schlie, D. Wille, S. Schulze, L. Cleophas, and I. Schaefer, "Detecting variability in matlab/simulink models: An industry-inspired technique and its evaluation," in *SPLC*, 2017, pp. 215–224.

[203] K. Czarnecki and M. Antkiewicz, "Mapping features to models: A template approach based on superimposed variants," in *GPCE*. Springer, 2005, pp. 422–437.

[204] G. Mussbacher, J. Araújo, A. Moreira, and D. Amyot, "Aourn-based modeling and analysis of software product lines," *Software Quality Journal*, vol. 20, no. 3, pp. 645–687, 2012.

[205] D. Strüber and S. Schulz, "A tool environment for managing families of model transformation rules," in *ICGT*. Springer, 2016, pp. 89–101.

[206] J.-P. Tolvanen and S. Kelly, "Creating domain-specific modeling languages for product lines," in *2011 15th International Software Product Line Conference*. IEEE, 2011, pp. 358–358.

[207] ——, "Describing variability with domain-specific languages and models," in *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A*, 2019, pp. 329–329.

[208] J. White, J. H. Hill, J. Gray, S. Tambe, A. S. Gokhale, and D. C. Schmidt, "Improving domain-specific language reuse with software product line techniques," *IEEE software*, vol. 26, no. 4, pp. 47–53, 2009.

[209] W. K. Assunção, S. R. Vergilio, and R. E. Lopez-Herrejon, "Discovering software architectures with search-based merge of uml model variants," in *ICSR*. Springer, 2017, pp. 95–111.

[210] J. Martinez, T. Ziadi, T. F. Bissyande, J. Klein, and Y. Le Traon, "Automating the extraction of model-based software product lines from model variants," in *ASE*. IEEE, 2015, pp. 396–406.

[211] J. Rubin and M. Chechik, "N-way model merging," in *FSE*. ACM, 2013, pp. 301–311.

[212] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin, "Model checking lots of systems: efficient verification of temporal properties in software product lines," in *ICSE*. ACM, 2010, pp. 335–344.

[213] K. Czarnecki and K. Pietroszek, "Verifying feature-based model templates against well-formedness ocl constraints," in *GPCE*. ACM, 2006, pp. 211–220.

[214] S. Peldszus, D. Strüber, and J. Jürjens, "Model-based security analysis of feature-oriented software product lines," in *GPCE*. ACM, 2018, pp. 93–106.

[215] J.-P. Tolvanen and S. Kelly, "How domain-specific modeling languages address variability in product line development: Investigation of 23 cases," in *SPLC*, 2019, pp. 24:1–24:9.

[216] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake, "Do background colors improve program comprehension in the #ifdef hell?" *Empirical Software Engineering*, vol. 18, no. 4, pp. 699–745, 2013.

[217] W. Fenske, S. Schulze, and G. Saake, "How preprocessor annotations (do not) affect maintainability: a case study on change-proneness," in *GPCE*, vol. 52, no. 12, 2017, pp. 77–90.

[218] C. Kästner, S. Apel, and K. Ostermann, "The road to feature modularity?" in *SPLC Companion*, 2011, p. 5.

[219] H. Spencer and G. Collyer, "#ifdef considered harmful, or portability experience with C news," in *USENIX*, 1992.

[220] S. Apel and C. Kästner, "An Overview of Feature-Oriented Software Development," *J. Object Techn.*, vol. 8, no. 5, pp. 49–84, 2009.

[221] D. Faitelson and S. S. Tyszberowicz, "UML diagram refinement (focusing on class- and use case diagrams)," in *ICSE*, 2017, pp. 735–745.

[222] P. P.-S. Chen, "The entity-relationship model—toward a unified view of data," *ACM transactions on database systems (TODS)*, vol. 1, no. 1, pp. 9–36, 1976.

[223] W. K. Assunção, S. R. Vergilio, and R. E. Lopez-Herrejon, "Automatic extraction of product line architecture and feature models from uml class diagram variants," *Information and Software Technology*, vol. 117, p. 106198, 2020.

[224] S. Ali, L. C. Briand, M. J.-u. Rehman, H. Asghar, M. Z. Z. Iqbal, and A. Nadeem, "A state-based approach to integration testing based on uml models," *Information and Software Technology*, vol. 49, no. 11-12, pp. 1087–1106, 2007.

[225] D. Gessenharter and M. Rauscher, "Code generation for uml 2 activity diagrams," in *ECMFA*, 2011.

[226] T. authors, "Online appendix to "effects of variability in models: A family of experiments"," 2021, https://sites.google.com/view/variabilitymodelingexperiment/home, https://zenodo.org/record/5578645.

[227] I. Vessey, "Cognitive fit: A theory-based analysis of the graphs versus tables literature," *Decision Sciences*, vol. 22, no. 2, pp. 219–240, 1991.

[228] Q. Ramadan, D. Strüber, M. Salnitri, J. Jürjens, V. Riediger, and S. Staab, "A semi-automated bpmn-based framework for detecting conflicts between security, data-minimization, and fairness requirements," *Software and Systems Modeling*, pp. 1–37, 2020.

[229] S. Alwidian and D. Amyot, "Union models: Support for efficient reasoning about model families over space and time," in *SAM*. Springer, 2019, pp. 200–218.

[230] T. Kühn and W. Cazzola, "Apples and oranges: comparing top-down and bottom-up language product lines," in *SPLC*, 2016, pp. 50–59.

[231] D. Clarke, M. Helvensteijn, and I. Schaefer, "Abstract delta modeling," *CW Reports*, 2010.

[232] C. Seidl, I. Schaefer, and U. Aßmann, "Deltaecore-a model-based delta language generation framework," *Modellierung 2014*, 2014.

[233] C. Pietsch, T. Kehrer, U. Kelter, D. Reuling, and M. Ohrndorf, "Sipl–a delta-based modeling framework for software product line engineering," in *ASE*. IEEE, 2015, pp. 852–857.

[234] I. Schaefer, "Variability modelling for model-driven development of software product lines," in *VaMoS*, 2010, pp. 85–92.

[235] P. Langer, T. Mayerhofer, M. Wimmer, and G. Kappel, "On the usage of uml: Initial results of analyzing open uml models," *Modellierung*, 2014.

[236] M. Petre, "Uml in practice," in *ICSE*. IEEE, 2013, pp. 722–731.

[237] V. Kulkarni, R. Venkatesh, and S. Reddy, "Generating enterprise applications from models," in *OOIS*. Springer, 2002, pp. 270–279.

[238] N. Moreno, P. Fraternali, and A. Vallecillo, "Webml modelling in uml," *IET software*, vol. 1, no. 3, pp. 67–80, 2007.

[239] S. Vaupel, G. Taentzer, J. P. Harries, R. Stroh, R. Gerlach, and M. Guckert, "Model-driven development of mobile applications allowing role-driven variants," in *MODELS*. Springer, 2014, pp. 1–17.

[240] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot, "Modisco: a generic and extensible framework for model driven reverse engineering," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 173–174.

[241] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[242] F. Chauvel and J.-M. Jézéquel, "Code generation from uml models with semantic variation points," in *MODELS*. Springer, 2005, pp. 54–68.

[243] E. Domı, B. Pérez, Á. L. Rubio *et al.*, "A systematic review of code generation proposals from state machine specifications," *Information and Software Technology*, vol. 54, no. 10, pp. 1045–1066, 2012.

[244] A. Reuys, S. Reis, E. Kamsties, and K. Pohl, "The scented method for testing software product lines," in *Software Product Lines*. Springer, 2006, pp. 479–520.

[245] I. Salman, A. T. Misirli, and N. Juristo, "Are students representatives of professionals in software engineering experiments?" in *ICSE*, 2015, pp. 666–676.

[246] U. Fahrenberg, M. Acher, A. Legay, and A. Wąsowski, "Sound merging and differencing for class diagrams," in *FASE*, 2014.

[247] B. Jones and M. G. Kenward, *Design and analysis of cross-over trials*. Chapman and Hall/CRC, 2003.

[248] M. M. Bergman, "The good, the bad, and the ugly in mixed methods research and design," 2011.

[249] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[250] J. C. De Winter and D. Dodou, "Five-point likert items: t test versus mann-whitney-wilcoxon," *Practical Assessment, Research & Evaluation*, vol. 15, no. 11, pp. 1–12, 2010.

[251] H. Abdi, "Bonferroni and šidák corrections for multiple comparisons," *Encyclopedia of measurement and statistics*, vol. 3, pp. 103–107, 2007.

[252] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[253] S. T. Karris, *Introduction to Simulink with engineering applications*. Orchard Publications, 2006.

[254] D. Whitney and D. M. Levi, "Visual crowding: A fundamental limit on conscious perception and object recognition," *Trends in cognitive sciences*, vol. 15, no. 4, pp. 160–168, 2011.

[255] S. Abrahão, F. Bourdeleau, B. Cheng, S. Kokaly, R. Paige, H. Stöerrle, and J. Whittle, "User experience for model-driven engineering: Challenges and future directions," in *MODELS*. IEEE, 2017, pp. 229–236.

[256] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger, "Bidirectional transformations: A cross-discipline perspective," in *ICMT*. Springer, 2009, pp. 260–283.

[257] B. Behringer, J. Palz, and T. Berger, "Peopl: Projectional editing of product lines," in *ICSE*. IEEE, 2017.

[258] K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged configuration through specialization and multilevel configuration of feature models," *Software process: improvement and practice*, vol. 10, no. 2, pp. 143–169, 2005.

[259] J. Schroeter, P. Mucha, M. Muth, K. Jugel, and M. Lochau, "Dynamic configuration management of cloud-based applications," in *SPLC*, 2012, pp. 171–178.

[260] T. Kaufmann, C. Manz, and T. Weyer, "Extending the SPES modeling framework for supporting role-specific variant management in the engineering process of embedded software," in *SE*, 2014, pp. 77–86.

[261] X. B. V. Chan, S. M. S. Goh, and N. C. Tan, "Subjects with colour vision deficiency in the community: what do primary care physicians need to know?" *Asia Pacific Family Medicine*, vol. 13, no. 1, p. 10, 2014.

[262] D. Gopher and R. Braune, "On the psychophysics of workload: Why bother with subjective measures?" *Human Factors*, vol. 26, no. 5, pp. 519–532, 1984.

[263] F. A. Aleixo, M. A. Freire, D. A. da Costa, E. C. Neto, and U. Kulesza, "A comparative study of compositional and annotative modelling approaches for software process lines," in *SBES*, 2012, pp. 51–60.

[264] A. R. Santos, I. do Carmo Machado, E. S. de Almeida, J. Siegmund, and S. Apel, "Comparing the influence of using feature-oriented programming and conditional compilation on comprehending feature-oriented software," *Empirical Software Engineering*, vol. 24, no. 3, pp. 1226–1258, 2019.

[265] T. Ziadi and J.-M. Jézéquel, "Software product line engineering with the uml: Deriving products," in *Software Product Lines*. Springer, 2006, pp. 557–588.

[266] D. Strüber and A. Anjorin, "Comparing reuse mechanisms for model transformation languages: Design for an empirical study." in *HuFaMo@ MoDELS*, 2016, pp. 27–32.