The Journal of Systems and Software 211 (2024) 112013

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

How can feature usage be tracked across product variants? Implicit Feedback in Software Product Lines☆

Oscar Díaz [a], Raul Medeiros [a,*], Mustafa Al-Hajjaji [b]

[a] University of the Basque Country (UPV/EHU), San Sebastián, Gipuzkoa, Spain
[b] pure-systems GmbH, Magdeburg, Germany

ARTICLE INFO

Dataset link: https://github.com/onekin/FEACKER, https://doi.org/10.5281/zenodo.8187116

Keywords:
Software Product Lines
Implicit feedback
Continuous development

ABSTRACT

Implicit feedback involves gathering information about software usage to grasp how and when the software is utilized. This study investigates the integration of implicit feedback mechanisms into Software Product Line Engineering (SPLE). While common in product-based development for identifying bugs, usability issues, and informing requirement prioritization, its adoption in SPLs has been limited due to the unique challenges posed by SPLE, such as the emphasis on Domain Engineering over Application Engineering, and the need for systematic reuse of shared assets. We propose a novel approach to incorporate feedback practices into Domain Engineering, thereby shifting the focus from individual product variants to the SPL platform, and specifically moving from product-based feedback to feature-based feedback. Based on a case study, we suggest that product derivation includes a second step that injects the trackers at the time of derivation, using a Feedback Model that complements the Configuration Model for feedback analysis.To test this approach, we introduce *FEACKER*, an extension to *pure::variants* as the variability manager. *FEACKER* injects trackers when the product variant is derived. The findings are validated through a Technology Acceptance Model (TAM) evaluation and a focus group discussion, providing insights into the feasibility, acceptance, and potential impact of platform-based feedback in SPLE. The results indicate agreement on the benefits of conducting feedback analysis at the platform level and the perception that *FEACKER* seamlessly extends the capabilities of *pure::variants*.

1. Introduction

Continuous deployment is considered an extension of Agile practices whereby the user feedback is extended beyond development by frequently releasing and deploying software to customers and continuously learning from their usage (Dakkak et al., 2021a). User feedback can be obtained either directly, e.g., through reviews (i.e., explicit feedback) or through tracking user interactions (i.e., implicit feedback). We focus on implicit feedback, i.e., the automatically collected information about software usage from which user behavior and preferences can be inferred (van Oordt and Guzman, 2021).

In product-based development, implicit feedback (hereafter just 'feedback') is achieved by embedding some specific code snippets: the trackers. These trackers monitor various user actions and behaviors without requiring direct input from the users themselves. The data collected might benefits developers and analysts alike. Developers can benefit from a more timely awareness of bugs, bad smells, or usability issues (Johanssen et al., 2019) while being motivated by their software's real value to real users (van Oordt and Guzman, 2021). Regarding the analysts, they resort to feedback for requirement prioritization (Wang et al., 2019; Johanssen et al., 2019) and requirement elicitation (Liang et al., 2015). Insights into which features are popular or underused can inform the development team on where to focus their improvement efforts or how to prioritize future development. No wonder feedback is catching on among product-based developers (Fitzgerald and Stol, 2017). Unfortunately, feedback has not received the same attention for Software Product Lines (SPLs) (Dakkak et al., 2021a).

Software Product Line Engineering (SPLE) aims to support the development of a whole family of software products (aka product variants) through systematic reuse of shared assets (aka the SPL platform) (Clements and Northrop, 2002; Pohl et al., 2005). To this end, SPLE distinguishes between two interrelated processes: Domain Engineering (DE), which builds the SPL platform, and Application Engineering (AE), which derives individual variants from the SPL platform.

☆ Editor: Laurence Duchien.

* Corresponding author.
E-mail addresses: oscar.diaz@ehu.eus (O. Díaz), raul.medeiros@ehu.eus (R. Medeiros), mustafa.alhajjaji@pure-systems.com (M. Al-Hajjaji).

https://doi.org/10.1016/j.jss.2024.112013
Received 28 February 2023; Received in revised form 6 November 2023; Accepted 23 February 2024
Available online 24 February 2024
0164-1212/

Traditionally, feedback is conducted by injecting 'usage trackers' directly into the product code (van Oordt and Guzman, 2021). However, a key characteristic of SPLE is to minimize coding efforts in AE and instead focus on DE. This principle should also apply to feedback. Therefore, we propose incorporating feedback practices into DE and, consequently, moving tracking practices from the SPL's products to the SPL's platform. This will result in feedback being based on the platform rather than individual products.

Unfortunately, experiences with feedback are rare for SPLs. The limited support for continuous deployment among variability managers (e.g., *pure::variants* (Beuche, 2019), *GEARS* (Krueger and Clements, 2018)) hinders this issue from being grounded on empirical evidence. On these premises, we introduce two research questions:

- How can platform-based feedback be incorporated into SPLE practices?
- How can variability manager tools support platform-based feedback?

We address these questions using *Action Design Research (ADR)* (Sein et al., 2011). ADR aims to solve the issues of a problematic situation, such as implicit feedback being conducted at the level of the product variant, by favoring a bottom-up approach where researchers join practitioners in their local problems/solutions, which are then argumentatively generalized. This bottom-up approach fits our setting where the problem arises from local practice, and this work aims to find a local solution that can next be generalized. In doing so, we contribute by:

- a proposal for framing feedback practices into SPLE: the IF4SPLE process (Section 4);
- a proof-of-concept of how IF4SPLE can be realized in *pure::variants* as the variability manager, and *Google Analytics* as the usage tracker: *FEACKER* (Section 6);
- a proof-of-value through a double evaluation: a Technology Acceptance Model (TAM) evaluation is first conducted among employees of *pure-systems GmbH* (n=8), followed by a focus group for the most divergent TAM questions (n=3) (Section 7);
- a generalization of both the problem instance, i.e., to what extent is *implicit feedback* a *problem* for SPL other than our case study, and the solution instance, i.e., to what extent is *FEACKER* a *solution* to implicit feedback in SPLs (Section 8).

We begin with background information on both SPLE and implicit feedback.

2. Background

2.1. Software product line engineering

Software Product Line Engineering (SPLE) is a systematic approach to software reuse that enables the efficient creation of diverse software products (aka variants) with shared components and features (Clements and Northrop, 2002; Pohl et al., 2005). SPLE enables organizations to achieve efficiency, flexibility, and cost savings by managing variability and leveraging commonalities across a family of related software products.

Artifact wise, SPLE handles two main assets: features and feature models. A feature represents a distinguishing functionality or characteristic that provides value to end-users or stakeholders (Apel et al., 2013). Features can be seen as building blocks that contribute to the overall functionality and differentiation of a product variant. A feature model, on the other hand, serves as a representation of the configurable options and relationships among features that scope the range of variants (aka the SPL portfolio). It defines the valid combinations and constraints of features, allowing for the systematic and controlled generation of specific variants (aka product derivation) (Galindo et al., 2019). Finally, features are realized through code, where different strategies for variability exist. The so-called core assets encompass the shared components, modules, libraries, and other reusable artifacts that conform to the SPL platform (Lindohf et al., 2021).

Process-wise, SPLE is usually divided into two interrelated processes: Domain Engineering (DE) and Application Engineering (AE) (Apel et al., 2013; Pohl et al., 2005). DE focuses on identifying and capturing the commonalities and variabilities of a specific application domain. It involves creating the feature model, defining the variability mechanisms, and developing reusable assets that form the foundation of the software product line ('developing for reuse'). In contrast, AE focuses on creating specific software products by selecting and configuring the desired set of features. This process involves mapping the requirements of individual customers or market segments to the feature model, determining the appropriate feature combinations, and generating the corresponding product configurations ('developing by reuse').

In terms of infrastructure, SPLE relies on variability managers, which are frameworks that assist in managing and controlling variabilities within a SPL. They provide support for creating and visualizing feature models, defining constraints and relationships among features, and facilitating the configuration and derivation of specific product variants. Variability managers differ in how variability is realized (Apel et al., 2013). First, annotation-based variability involves using annotations or metadata to specify and control variations within the system. Annotations are added to the source code or other artifacts to indicate the presence of certain features or to enable or disable specific functionalities. Second, component-based variability relies on component composition, where components are assembled and interconnected according to specified feature configurations. Third, feature-oriented programming (FOP) is a programming paradigm that provides language constructs and techniques designed specifically to handle variabilities at the code level. FOP involves explicitly representing features as first-class entities in the code, allowing for modularization and encapsulation of feature-specific functionalities. Finally, aspect-oriented programming (AOP) is often used in product-based development to address cross-cutting concerns such as logging, error handling, or security. In AOP, crosscuts are encapsulated as reusable aspects that can be selectively applied based on specified feature configurations. This work is specifically about a crosscut: implicit feedback and its incorporation in SPLE. Next, we will briefly discuss the relevance of implicit feedback.

2.2. Implicit feedback

Feedback refers to the information, insights, and opinions provided by stakeholders, users, or other relevant parties regarding the software system or development process. Specifically, the literature distinguishes two customer feedback types: explicit and implicit. The first refers to what users directly say or write about a software (e.g., tweets, reviews written in an app store, etc.) (van Oordt and Guzman, 2021; Johanssen et al., 2019). By contrast, implicit feedback automatically collects data about the software usage or execution from which user preferences and trends can be inferred (Maalej et al., 2009; van Oordt and Guzman, 2021). In short, explicit feedback is how users see the software, while implicit feedback depicts how they actually use it. We focus on implicit feedback.

Feedback plays a crucial role in identifying issues, evaluating performance, and improving the quality and usability of software. Recently, there has been increased interest in feedback mechanisms due to the industry's adoption of agile and continuous methods. These methods inherently include customers' feedback in the analysis loop of the development process (Schön et al., 2017; Johanssen et al., 2019). Specifically, uses of implicit feedback in product-based development include:

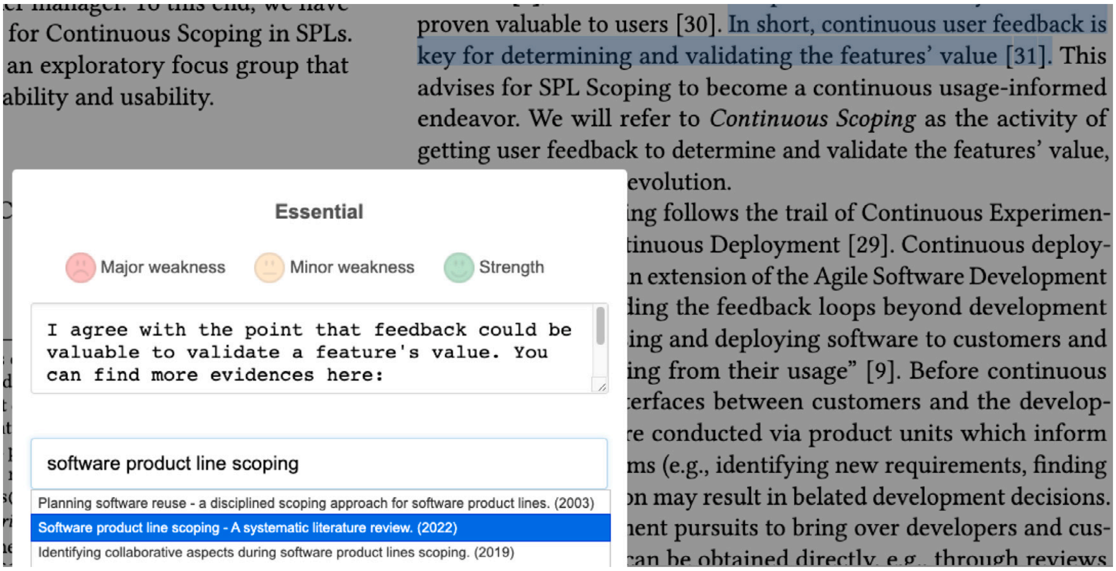

Fig. 1. The *Commenting* feature. Users can not only add notes but also search for bibliographical references in Google Scholar.

- *Requirement Prioritization.* Based on actual feature usage, managers prioritize which requirement implementation or maintenance should be performed first (Johanssen et al., 2019; Hoffmann et al., 2020; van Oordt and Guzman, 2021; Johanssen et al., 2018).
- *Requirement Elicitation.* Discovery of new requirements by analyzing the different usage paths of the users (Hoffmann et al., 2020; Oriol et al., 2018)
- *Optimization.* Improvement of different quality indicators (Martínez-Fernández et al., 2019; Johanssen et al., 2019) or optimization of functionalities to be easier and faster for the users to use them (van Oordt and Guzman, 2021).
- *Bug Awareness.* Early identification and fixing of bugs by capturing them through crash reports and raised errors in the logs (van Oordt and Guzman, 2021; Johanssen et al., 2019; Olsson and Bosch, 2013).
- *User Understanding.* Understanding how users interact with the software (e.g., analyzing how they adapt to new functionalities) (van Oordt and Guzman, 2021; Johanssen et al., 2019).

SPL teams also need to prioritize requirements, optimize code, and understand users. However, the key difference lies in where the decision-making process occurs: at the platform level rather than the product level. In SPL, the prioritization of requirements and code optimization typically happens at the platform level, rather than being based on individual products. Although application engineers can offer information about specific products, such as usage or bugs, the ultimate decision on which code to optimize or which requirement to prioritize often rests with the Control Board. Therefore, we propose harmonizing implicit feedback with SPLE practices by shifting it to the platform level.

3. Practical case: Implicit feedback at *WACline*

This section introduces the case that triggered this research: *WACline*, an SPL in the Web Annotation domain (Medina et al., 2022). Web Annotations conform to a digital layer of interactivity on the Web, allowing users to add comments, corrections, and highlights to various types of digital content. They can be applied to web pages, ebooks, videos, images, and more, and can be linked, shared, traced back to their source, and searched (Sanderson, 2016). The practice of annotation (i.e., how annotations are produced, visualized, or shared) is very heterogeneous. Different annotation practices can be found in Social Sciences and Humanities (Caria and Mathiak, 2020), Journalism investigation (Rehm et al., 2019) or Medical and Biological Sciences (Rehm et al., 2015), just to name a few. This variability sustains the use of an SPL approach for the development of annotation tools.

3.1. The running example

The domain. *WACline* supports the fast and robust creation of web browser extensions that allow users to create Web Annotations on any kind of web resource. The current version of *WACline* boasts a total of 86 distinct features, categorized as follows: 23 mandatory features, 11 alternative features, 12 features with an unrestricted choice (aka or), and 40 optional features (see Fig. 2). The actual SPL portfolio consists of eight production-ready variants, yet the feature model has the configurable capacity to generate over 38 million variants.[1] As the running example, Fig. 1 shows the case of the *Commenting* feature. This feature allows adding a comment to highlighted text or to a particular element (e.g., an image). In addition, it allows reviewers to query Google Scholar directly without opening a new browser tab.

The implementation. *WACline* resorts to annotation-based variability. This involves employing pre-processor directives to determine when a block of code should be included in the final product variant, based on the presence or absence of a feature selection during configuration. Using *pure::variants* as the variability manager, *WACline* variation points in the source code are denoted through *pure::variants*' native pre-processor directives that start with an opening directive *//PVSCL:IFCOND* and end with a closing directive *//PVSCL:-ENDCOND*. Fig. 3 (left) provides an example with the *//PVSCL:IFCOND* directives (aka *#ifdef* block) framing part of the code-bases of *Commenting* and *Replying* features.[2]

The problem. *WACline* delivers web products. Web applications are one of the earlier adopters of implicit feedback. This is achieved with the help of web analytic services such as Google Analytics (GA)

[1] The number of possible variant configurations was calculated using 'show metrics' functionality offered by *pure::variants*.

[2] It is crucial to distinguish between the act of annotating code with pre-processors and the concept of Web Annotation. The former is a technical implementation of variability within the code, whereas the latter refers to functional capabilities.

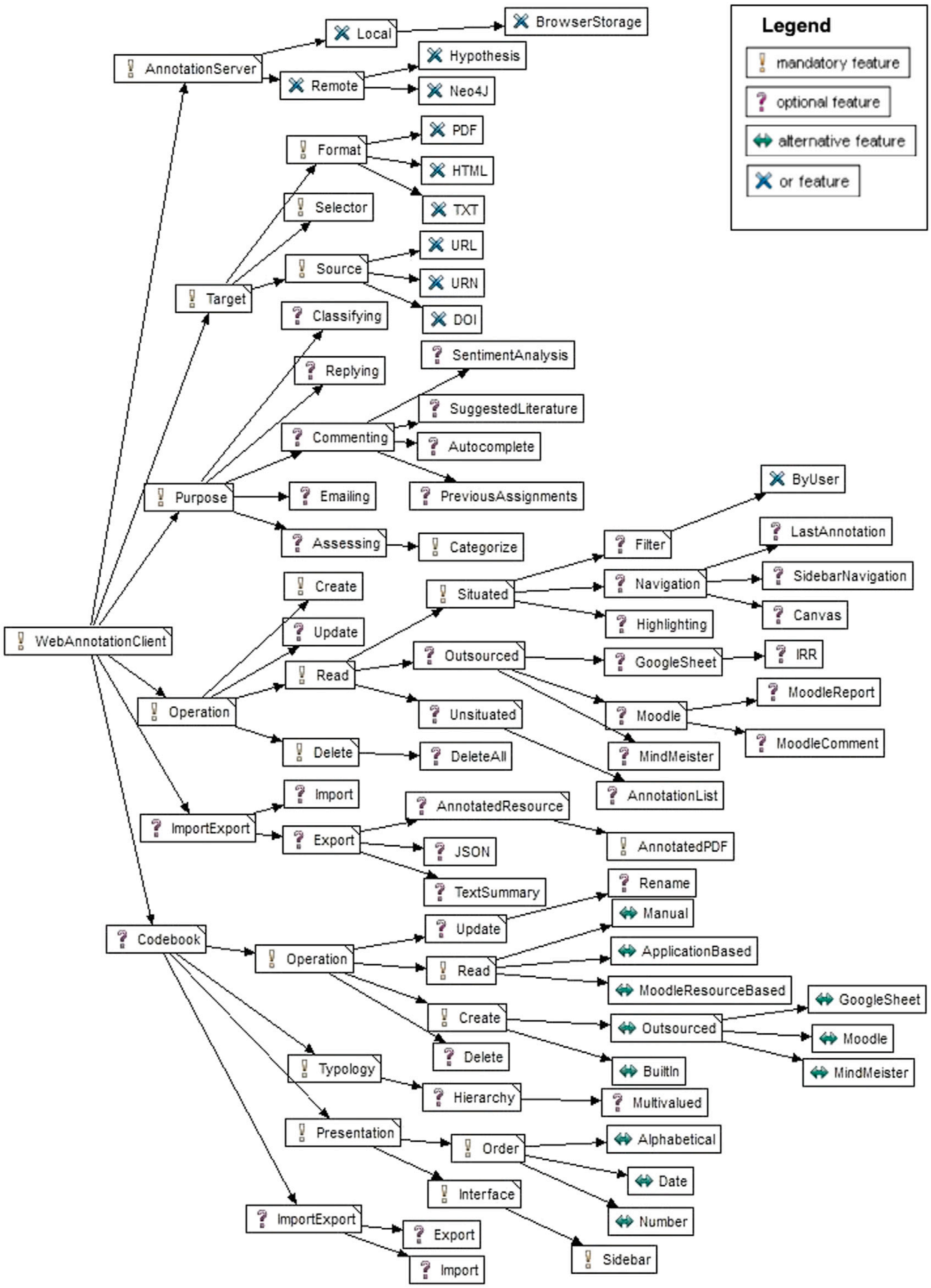

Fig. 2. *WACline*'s feature model.

(W3Techs, 2022). GA tracks and reports website usage. It provides insights into how visitors interact with a website, including the number of visitors, their behavior, and their demographics. Operationally, usage tracking is achieved by inlaying tracking instructions (known as *hits* in the GA parlance). Fig. 3 (right) displays an example of *Commenting* once incorporated into a *WACproduct* (notice that the *#ifdef* block is no longer there). *WACproduct* is now 'feedback-minded', i.e., each execution of this code populates the feedback log with an event occurrence. Those hits run in the web browser of each client, collecting data and sending it to the feedback log. Eventually, this feedback log is consulted through appropriate dashboards (Google, 2016).

```
openForm (annotation) {
  Form.showForm(annotation, (err, annotation) => {
    // PVSCL:IFCOND(Commenting, LINE)
    if (err) {
      Alerts.errorAlert({ text: 'Unexpected error'})
    } else {
      LanguageUtils.dispatchCustomEvent(
        Events.createComment, {
         annotation: annotation
        })
    }
   // PVSCL:ENDCOND
   // PVSCL:IFCOND(Replying, LINE)
     LanguageUtils.dispatchCustomEvent(
       Events.createReplying, {
         annotation: annotation
       })
  // PVSCL:ENDCOND
  })
}
```

```
openForm (annotation) {
  Form.showForm(annotation, (err, annotation) => {
    if (err) {
      Alerts.errorAlert({ text: 'Unexpected error'})
    } else {
      LanguageUtils.dispatchCustomEvent(
        Events.createComment, {
         annotation: annotation
        })
      ga('send',{
        hitType: 'event',
        eventCategory: 'Commenting',
        eventAction:'commentingEnacted',
        variant:'WACproduct'
      })
    }
  })
}
```

Fig. 3. *Platform* code (with pre-compilation directive) vs Feedback-minded *variant* code (i.e., each execution of this code populates the feedback log with an event occurrence described along with the 'eventCategory' (e.g., *Commenting*), the 'eventAction' (e.g., *CommentingEnactment*) and the 'variant' (i.e., the product which raises the event).

However, *WACproduct* is not a standalone product, but a product variant derived from the *WACline* platform. Realizing implicit feedback in the conventional way (i.e., at the variant level) is problematic. While providing tracking code at the variant level, *WACline* engineers were aware of eventual risks in terms of delays, partiality (i.e., conducting changes for some variants but not all), and chances of suffering 'merge hell' in the Git repository. The following paragraphs delve into these issues that ultimately motivated this work.

Risk of delays: Continuous deployment brings developers and customers closer, reducing the latency in development decisions often associated with the traditional model where product units serve as the interface between customers and development teams. SPLE introduces an additional layer of indirection between users and code developers. As code developers, domain engineers might not have direct access to the users but need to rely on the application engineers. This additional layer can lead to delays in receiving user feedback, which slows down the decision-making.

Risk of partiality: In their eagerness to promptly understand their customers' needs, application engineers might be tempted to conduct feedback analysis locally. This means that they directly inject hints into their product variants, favoring changes at the product level. However, this approach overlooks the implications for the SPL platform as a whole.

Risk of 'merge hell': When kept in a Git-like repository, the SPL platform stands for the *main* while changes made to the product variant are realized as branches. These variant branches need to be eventually merged back into the *main*. Missing to reintegrate these product variants back to the *main* risks SPLE becoming clone&own development (Díaz et al., 2022). If GA hits are added at the variant level, then each AE team will inject them for their purposes. However, this way of realizing feedback increases the likelihood of encountering merge conflicts while merging variant branches back into the platform's *main* branch.

Alternatively, engineers at *WACline* attempted to define feedback as a feature, but it did not work either. Usage feedback is not a distinctive user-visible aspect of a variant, but rather a measurement for decision-making that might involve tracking distinct single features or feature combinations (e.g., whether F1 and F2 are used together). Feedback directives can be considered crosscuts upon the SPL's feature model. In *WACline*, this situation highlighted two significant inconsistencies when conducting feedback at the variant level:

- a conceptual mismatch (the tracking subject). SPLs reason in terms of *features*, while GA (and other analytic services) considers products as the unit of tracking,
- a process mismatch (the scope). SPLs set the analysis for a collection of product variants (i.e., the SPL portfolio), while GA is thought to track a single product.

These observations led to the initiative of integrating implicit feedback into existing variability managers to make feedback part of domain engineering. To this end, we applied an approach based on Action Design Research (ADR) (Sein et al., 2011). ADR involves collaboration between researchers and practitioners throughout all stages, not just evaluation. This ensures that the resulting IT artifact is both theoretically grounded and practically applicable to the organization's needs. ADR is also suitable for problems that require iterative development and continuous refinement based on stakeholder interactions. Our setting fits this description.

4. Feedback-minded variability managers

SPLE relies on variability managers, i.e., frameworks that assist in managing and controlling variabilities within a SPL. If feedback is added as an additional task in SPLE, how can variability managers assist? How is the SPLE process altered? How does feedback analysis intermingle with the existing SPLE tasks? This section makes a case for Implicit Feedback to be incorporated into SPLE practices: the IF4SPLE process.

Using Apel et al. process as the base (Apel et al., 2013), Fig. 4 outlines the IF4SPLE process. Specifically:

- Domain engineering now includes *Feedback Specification* (i.e., the description of the data to be collected in the feedback log), and *Feedback Analysis* (i.e., the study of the data in the feedback log),
- Application engineering now incorporates *Feedback Transformation*, which involves embedding feedback tracking instructions during variant derivation, as well as *Feedback Gathering*, which entails collecting feature usage during the execution of the product variant.

The following subsections delve into these tasks.

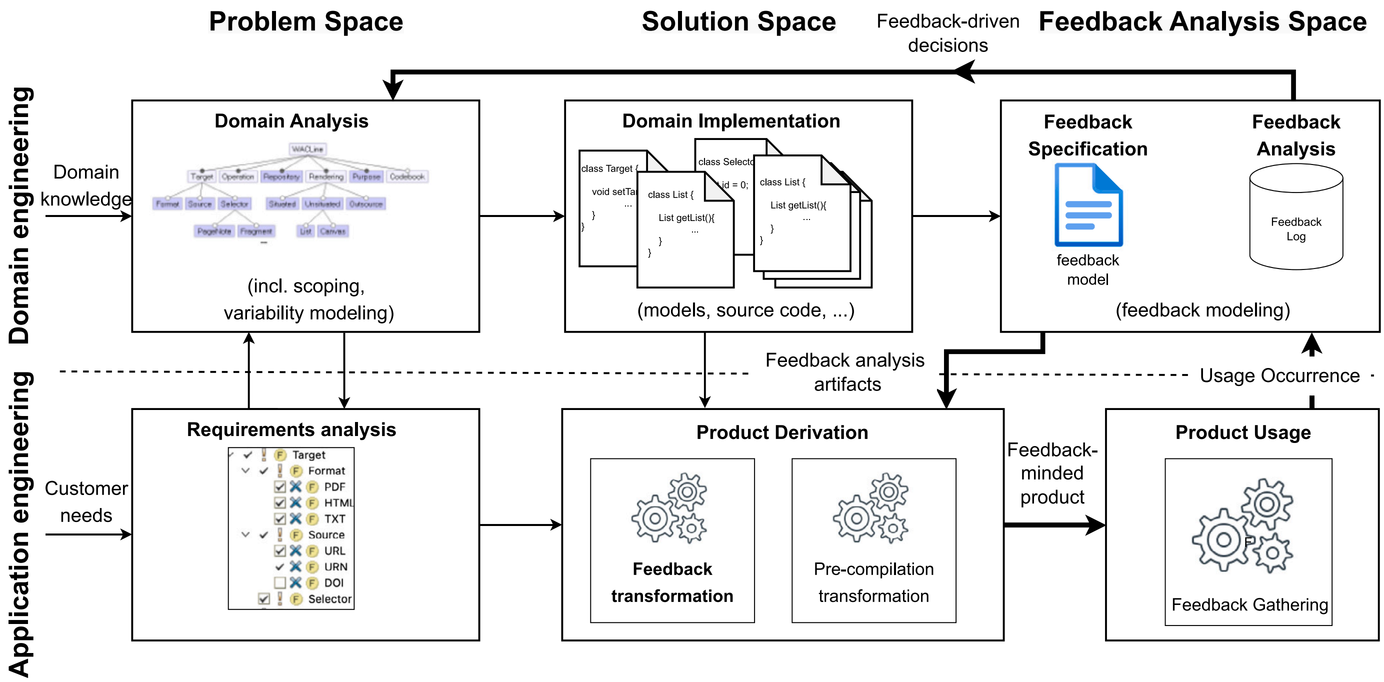

Fig. 4. The IF4SPLE process for leveraging SPLE with Feedback Analysis. The bold line denotes the feedback workflow: analysis-specification-coding-dataGathering-analysis.

4.1. Feedback specification

As in product-based development, feature usage can serve as a primary source of information for various decision-making issues throughout the SPL life-span. The Goal-Question-Metric (GQM) approach is a well-known framework for guiding goal-oriented measurements within a software organization (Basili and Rombach, 1988). By establishing specific goals and identifying key performance indicators, the GQM approach offers a structured method for decision-making. We then consider Feedback Specification a GQM practice with a focus on metrics related to feature usage. The rest of this section is structured along these three elements (i.e. the goal, the question, and the metrics).

The Goal. Basili and Rombach (1988) introduce the following template for goal description:

> Analyze <*target*> with the purpose of <*purpose*> with respect to <*quality focus*> from the point of view of the <*perspective*> in the context of <*context*>

In product-based development, the 'target' is a single product. However, SPL heavily rests on features as a significant construct throughout the SPLE process. Accordingly, it is natural to introduce features as the target of feedback analysis. After all, chances are that agendas, organizational units or release schedules are set regarding features (Hellebrand et al., 2017; Wnuk et al., 2009). Hence, features become the target.

Yet, we observe that for some scenarios, single features are not enough. Although tracking the usage of individual features was common, it was also common to consider particular feature combinations (e.g., as described in the pre-compilation directives that form the *#ifdef* blocks). Hence, the codebase to be analyzed (i.e., the *#ifdef* blocks) may be filtered based on whether they exhibit a single feature or a combination of features in their pre-compilation directive. This will identify 'the slice of the SPL code-base' that represents the target.

This code slice (the target) might need to be analyzed for various purposes such as refactoring or testing. The analysis should be done with regard to specific audiences, such as domain engineers, testers, product units, and application engineers. Setting a parallel with how feature changes are managed, a Control Board might be responsible for this, but it may vary depending on the context. In some organizations, UX designers or even some product engineers may be allowed to define their own feedback requirements.

We introduce some examples heavily based on the WACline experience. Consider the *Commenting* feature as the target (see Fig. 1). Possible purposes include:

- Feature scoping. Besides adding notes, *Commenting* allows for bibliographical references to be added by incorporating a Google-Scholar lookup functionality. Analysts might want to assess the extent to which this functionality is utilized. The rationale is that if this particular utility is seldom used in comparison to the usage of *Commenting*, then it might be better to detach it from *Commenting* and offer it as a separate configuration option.
- Feature optimization. *Commenting* is made available in two ways: 'double click' and 'right click'. Engineers are concerned that this dual choice may confuse users and want to determine which approach is the most popular for accessing this feature.

If the Control Board conducts this study in the context of planning the next *WACline* release, the resulting GQM goal is left as follows:

> Analyze *Commenting* with the purpose of *scoping & optimization* with respect to *usage* from the point of view of the *Control Board* in the context of *the planning for the next release of WACline*

So far, the example assumes the context to be the SPL platform as a whole. Though the target is limited to the *Commenting*'s code-base (i.e., the *#ifdef* blocks involving *Commenting*), the context is the entire SPL portfolio, i.e., track *Commenting* no matter the variant in which this feature is deployed.

However, we also came across scenarios where the analyst was interested in the usage of a feature but for a particular set of variants. Back to the running example, *Commenting* adds menus and widgets to the Web interface. If *Commenting* is utilized with other GUI-intensive features, such as *Highlighting*, the combined effect may lead to cluttered interfaces that can potentially discourage users. However, the feature model does not explicitly state a feature dependency between *Commenting* and *Highlighting* since there is no functional dependency. Yet, avoiding cluttered interfaces is a common purpose during usability evaluation. Monitoring this purpose then requires the tracking of *Commenting* but only in the presence of *Highlighting*. *Highlighting* is a feature that serves to frame the context of interest. In this case, the goal would look as follows:

> Analyze *Commenting* with the purpose of *usability evaluation* with respect to *usage* from the point of view of the *UX designer* in the context of the *Highlighting* realization

In the above example, *Commenting* is the target, while *Highlighting* serves to scope the part of the SPL portfolio that needs to be tracked (i.e., the context). Specifically, it is used to track *Commenting* only for those products that have *Highlighting* as part of their configuration.

The Questions. Questions help to break down the high-level goals into more specific and actionable inquiries. Hence, questions should be measurable, meaningful, and relevant to the goal at hand Basili and Rombach (1988). Back to our running example, previous goals on *Commenting* could be addressed through two questions, namely:

- Q1: How is *Commenting* used? *Commenting* permits Google-Scholar lookup from within. The Control Board wonders about the appropriateness of detaching the Google-Scholar lookup functionality from *Commenting* and offering it as a configuration option within *Commenting*.
- Q2: How is *Commenting* enacted? This feature offers two enactment modes: *doubleClickEnactment* or *rightClickEnactment*. Assessing the popularity of each of these modes helps to determine if they are alternatives or if any of them is residual, and hence, a candidate to step out in the next release.

The Metrics. Metrics refer to quantifiable measures that are used to collect data and evaluate progress towards achieving the defined goals (Basili and Rombach, 1988). For our running example, three possible metrics include

- M1: What is the enactment ratio of *Google-ScholarLookup* w.r.t. the enactment of *Commenting*?
- M2: What is the enactment ratio of *doubleClickEnactment* w.r.t. the number of times *Commenting* is enacted?
- M3: What is the enactment ratio of *rightClickEnactment* w.r.t. the number of times *Commenting* is enacted?

Specifically, the metric calculation formula for M1 will look as follows:

$$M1 = \frac{\#googleScholarLookupHit}{\#doubleClickEnactmentHit + \#rightClickEnactmentHit}$$

Here, *GoogleScholarLookUpHit*, *doubleClickEnactmentHit*, or *rightClickEnactmentHit* are variables to be retrieved from the feedback log (see later). This feedback log is fed with hits, i.e., events generated at run-time from the product variants that need to be tracked. To signal hits at run-time, usage trackers need to be located in the code. Drawing from the AOP literature (Kiczales et al., 1997), we refer to these locations as 'pointcuts'. The same hit may be signaled from different pointcuts. As we will see later, *DoubleClickEnactmentHit* accounts for two pointcuts: one for each of the code points where *Commenting* can be enacted with a double-click. The need for different pointcuts arises from the varying implementation of *Commenting*, depending on whether the comment is applied to highlighted text or to an element, such as an image.

So far, we have introduced the different pieces of information to address feedback requests, structured according to GQM practices. Fig. 5 depicts the distinct constructs of the Feedback (meta)Model. The model reflects two main entities: Goal and Hit. A goal is a collection of questions that, in turn, aggregate metrics. On the other hand, a hit aggregates a set of pointcuts. Finally, Goal and Hit have an (indirect) M:N relationship through Metric. A metric contains a formula that may refer to different hits, while a hit may be referred to in formulas held by different metrics.

Section 6.1 will introduce the concrete syntax for this model using YAML.

4.2. Feedback transformation

Traditionally, the SPL platform includes a configuration mechanism that allows users to select, customize, and configure the desired features and options for a specific product variant. The configuration mechanism ensures that the derived products are consistent, valid, and meet the requirements of different customers. The aim is to reduce, if not eliminate, updates to the product code-base once it has been generated. This includes any tracking instructions code aimed at gathering feedback. To achieve this goal, usage hits should also be generated instead of being directly coded into the product code-base.

Rather than cluttering the SPL code-base, 'separation of concerns' suggests that feedback-specific code should be added dynamically during the derivation of the product. In this approach, feedback needs are independently specified as a Feedback Model. As a result, the product code is derived from two inputs: (1) the configuration model that captures the chosen set of feature adaptations specific to a particular product variant, and (2), the feedback model that captures feedback analysis goals according to the Feature (meta)Model constructs (see Fig. 6). The outcome is a product variant with embedded tracking hits.

4.3. Feedback gathering

Feedback gathering refers to capturing usage data from users (Johanssen et al., 2019). Usage data should meet specific quality properties (Redman, 2013), namely accuracy (i.e., tracking hits should be error-free and reflect real-world values), completeness (i.e., tracking hits should collect the information specified in the Feedback Model), consistency (i.e., data gathered should be consistent among the different products) and, timeliness (i.e., data should always be up-to-date and reflect the real use of the platform). Thus, gathering feedback goes beyond a programming issue to also include considerations of data governance. The question is what SPLE brings to account for these quality factors, namely:

- accuracy. In terms of SPLE, setting the target in terms of features provides a declarative way to define the scope of the feedback analysis. Instead of naming the specific products to be tracked (extensional approach), SPLs use the concept of feature to scope the relevant variants. However, it should be noted that faults in the feedback specification will have a larger impact as faults will affect a larger set of product variants.
- completeness. Assessing completeness in gathering implicit feedback data faces the challenge of capturing a wide range of product variants. Gathering data from all deployed variants might not be practical due to the large number of variants and varying user adoption rates. To address this challenge, efficient sampling techniques, and statistical analysis should be utilized. Strategic sampling helps select a representative subset of product variants and user interactions, prioritizing diverse segments and relevant features.
- consistency. In terms of SPLE, the derivative approach ensures that the feedback code (i.e., hits) is generated consistently regardless of the variants in which it is injected.

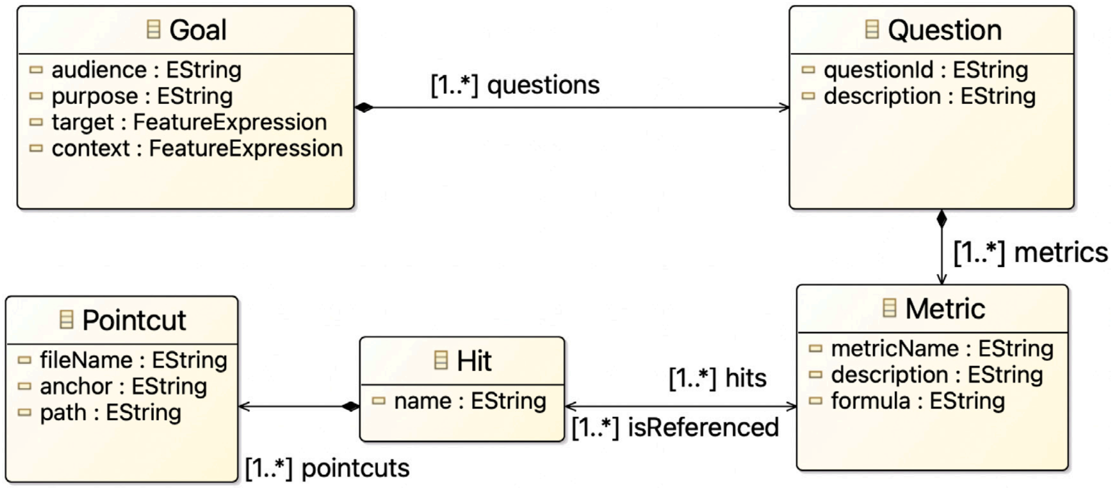

Fig. 5. The Feedback (meta)Model, which conforms to the Ecore metamodel. Both *Goal's* context and target can be described through feature expressions as in *#ifdef* directives (e.g. *Commenting* AND *Highlighting*). *Questions* break down goals into measurable inquiries that are measured through *Metrics*. Finally, Metric's formulas are calculated based on *Hits* occurrences raised in different code locations (*Pointcuts*).

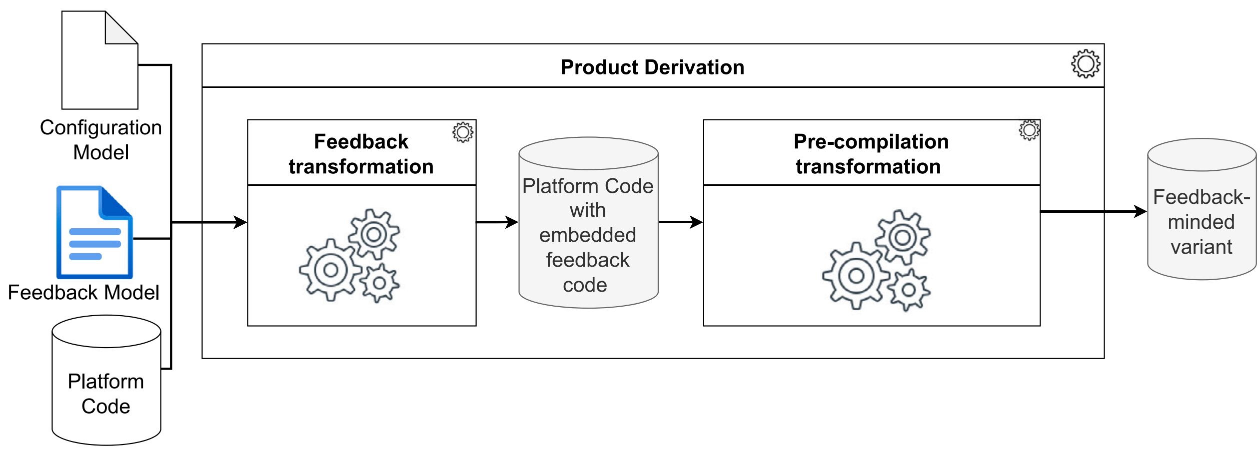

Fig. 6. Two-step Product Derivation. First, the Feedback Transformation injects the tracking hit along with the Feedback Model. Second, the Pre-compilation Transformation filters out those *#ifdef* blocks that do not meet the configuration model. The result is a feedback-minded product variant.

- timeliness. Feedback should evolve using the derivative approach, which speeds up changes during analysis. Instead of revising individual variants, changes are made at the Feedback Model. However, the model's inclusion of pointcuts tightly links the feedback specification to the feature code. This may necessitate changes in the Feedback Model when there are changes in the codebase. We might need to consider separating concerns, although we currently lack sufficient evidence to evaluate the pros and cons of detaching pointcuts from the feedback specification.

4.4. Feedback analysis

The Feedback Model captures feedback needs in terms of the features at the SPL level. Following the directives of the Feedback Model, feedback transformation makes product variants feedback-minded, that is, the execution will populate the feedback log. Each entry in the feedback log represents a usage hit. These hits are the smallest units of analysis, which are then aggregated to calculate the GQM's metrics. These metrics help answer decision-making questions. Therefore, variability managers should also offer means to access the feedback log and provide appropriate visualizations. (e.g., dashboards (López et al., 2021)). The question arises whether existing tooling can be used. For instance, *Google Analytics* offers dashboards for website owners to analyze trends and patterns in their visitors' engagement with the website (Plaza, 2011). However, SPL scenarios differ in two key aspects. Firstly, the focus of the tracking is not a single application but rather (a subset of) the SPL portfolio. Secondly, features become the central unit of analysis. Thus, dashboards should be designed to accommodate these unique characteristics.

To conclude, this section examines the potential impact of introducing feedback in SPLE (see Fig. 4):

- From an organizational perspective, engineers need to specify the Feedback Model. The scope of this specification is not limited to a product variant but extends to a subpart of the SPL platform. Therefore, a Control Board should be available to supervise and manage changes. When multiple change requests are received, the Control Board prioritizes them based on their urgency, importance, and potential impact. Both domain engineers and application engineers may participate.
- From a technical perspective, it is advisable to use variability management frameworks to simplify the specification of the Feedback Model using the existing feature model and core assets. This requires the use of editors, validators, transformers, and dashboards that are specifically designed for feedback specification and realization in SPLs.

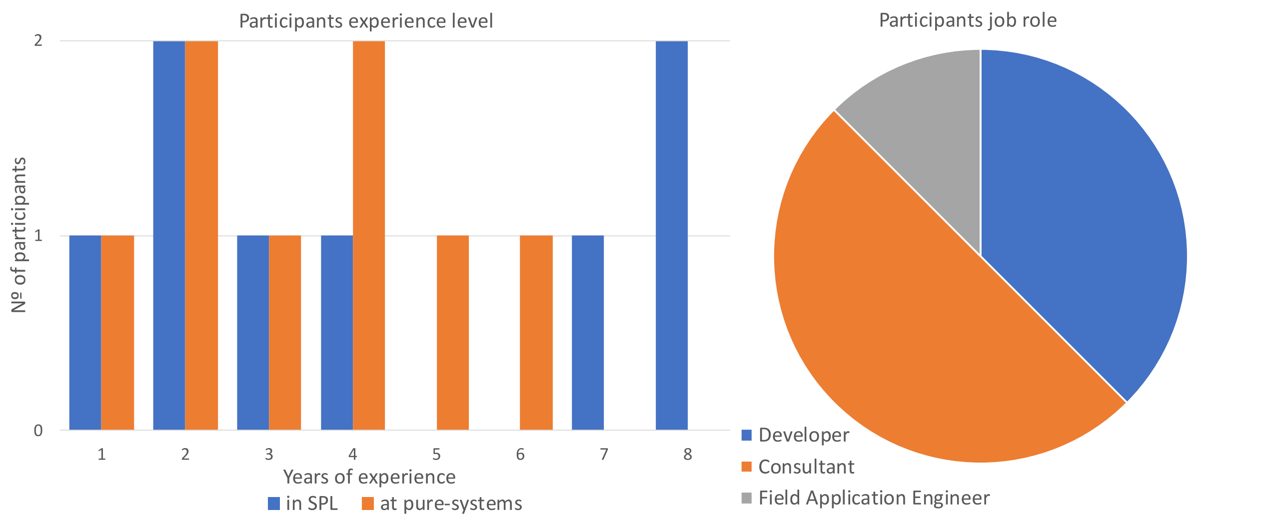

Fig. 7. Participants demographics: SPL experience & role played.

5. Validating the requirements

The previous subsection introduced a way to support implicit feedback in SPLE, named IF4SPLE. This proposal was derived from our experience handling implicit feedback in *WACline*. To improve external validity, this section conducts a survey on the extent IF4SPLE accounts for *coherence* (i.e., the quality of forming a unified whole with traditional SPLE practices) and *consistency* (i.e., the quality of seamlessly fitting with prior roles in SPLE). The next paragraphs describe the evaluation of the IF4SPLE principles.

Goal. The purpose of this study is to *assess* the coherence and consistency of the IF4SPLE principles for *seamlessly introducing implicit feedback into SPLE practices* from the point of view of *annotation-based SPL practitioners* in the context of *pure-systems GmbH*[3] and annotation-based SPLs.

Participants. We resorted to *pure-systems GmbH*, which is a leading provider of software product lines and variant management engineering tools, including *pure::variants*. The suitability of this company is twofold. First, *pure-systems GmbH*, as a leading SPL consultant company, is well placed to assess whether the IF4SPLE process suits well with the current SPLE practices. Second, *pure-systems GmbH* acts as a consultant for a broad range of companies, and hence, faces different requirements and domains. Hence, its participation will provide a high variation of settings in which to support generalization.

Design. The experiment was conducted during the monthly seminar regularly held at *pure-systems GmbH*. The seminar lasted for 90 min and aimed to introduce the participants to the importance of implicit feedback in product-based development and its potential benefits in the context of SPLE. During the seminar, the participants were also introduced to IF4SPLE. Once the seminar was over, participants were asked to fill out the questionnaire anonymously (see next paragraph). Fifteen employees attended the seminar, yet only eight completed all the questions of the survey. For these eight employees, demographic data is displayed in Fig. 7.

Instrument. To better profile what is meant by ‘a seamless introduction’, we characterize this adverb in terms of *coherence* (i.e., the quality of forming a unified whole with traditional SPLE practices) and *consistency* (i.e., the quality of seamlessly fitting with prior roles in SPLE). We resort to a questionnaire created on purpose where items are assessed through a five-point Likert.[4] Each item is concerned with one of the tasks introduced by IF4SPLE (i.e., Feedback Specification, Feedback Transformation, and so on).

While coherence considers the process, consistency refers to the roles played by participants in an SPL organization. Specifically, we consider two roles (i.e., DE vs AE). The rationale reads as Implicit Feedback might be conducted as part of DE with the participation (or not) of application developers. And the other way around, it is possible for Implicit Feedback to be defined for a set of products without implying the whole platform. Coherence wise, it is possible for practitioners to have different interpretations of SPL practices and the extent to which they understand the questionnaire. We tried to mitigate this risk by involving practitioners from the same company (who are exposed to the same practices) and conducting a 90-minute seminar to introduce the terminology. However, it should be noted that while the seminar may have helped standardize the understanding of implicit feedback, it assumes a certain common understanding of SPL concepts such as domain engineering or features. Figs. 8 and 9 show the results of IF4SPLE’s coherence and consistency, respectively.

When it comes to assessing the internal consistency, or reliability, of questionnaires, Cronbach’s alpha coefficient measure is used (Tavakol and Dennick, 2011). It helps determine whether a collection of questions consistently measures the same characteristic. Cronbach’s alpha quantifies the level of agreement on a standardized 0 to 1 scale. Higher values indicate higher agreement between items. Thus, high Cronbach’s alpha values indicate that response values for each participant across a set of questions are consistent. On these grounds, we calculate the Cronbach’s alpha values of the questionnaire items for coherence and consistency, leading to a result of 0.63, and 0.8 alpha values, respectively. In the case of consistency, we reversed the results of questions CON3 and CON5 as they favor AE to conduct implicit feedback tasks. An acceptable degree of α reliability is defined as 0.6 to 0.7, and a very good level is defined as 0.8 or greater (Ursachi et al., 2015). Therefore, we can consider the questionnaire reliable enough. It should be noted that Cronbach’s alpha index only conveys whether the answers are inherently coherent but it does not capture that the participants contradicted each other.

Results. **Coherence.** Participants generally believe IF4SPLE is a natural fit for current SPLE practices (question COH1). Feedback Specification (question COH2) and Feedback Analysis (question COH5) were

[3] https://www.pure-systems.com/

[4] A replication package with the questionnaire and the response data is available: https://doi.org/10.5281/zenodo.8187116

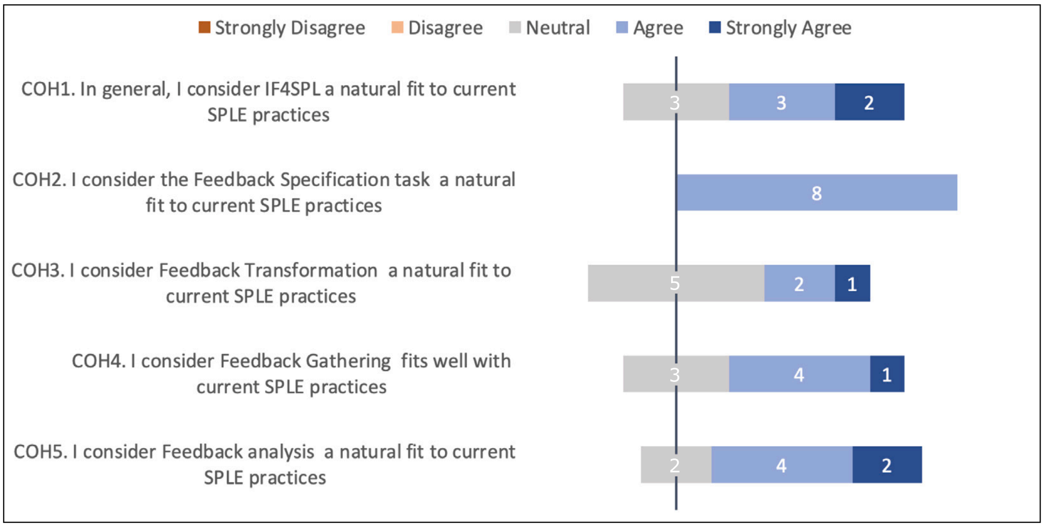

Fig. 8. Assessing coherence for IF4SPL.

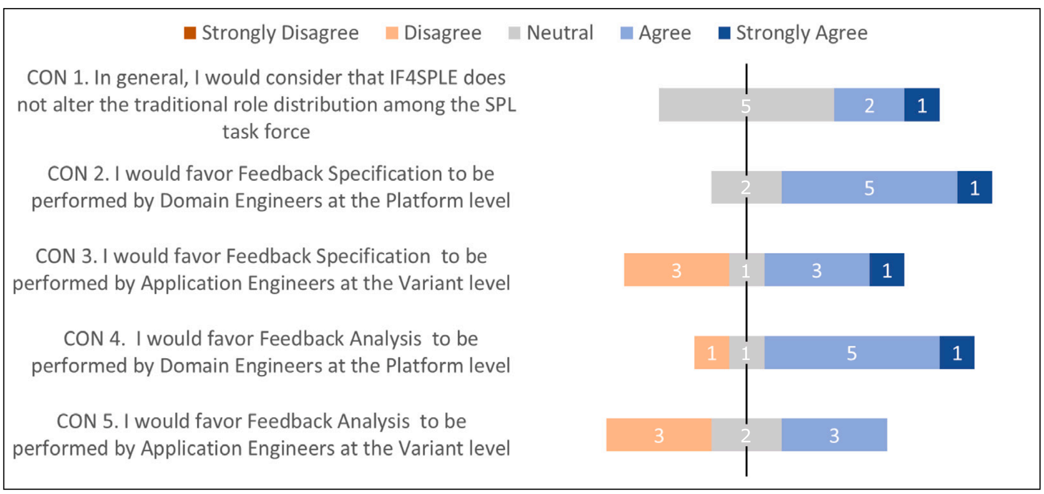

Fig. 9. Assessing consistency for IF4SPL.

the tasks that unanimous agreement reached about being conducted at the level of DE, hence moving Implicit Feedback from being a product concern to being a platform concern. Surprisingly, while positive, feedback transformation (i.e., injecting feedback concerns when the product is generated mimicking traditional techniques based on pre-compilation directives) did not meet our expectations, with most participants ranking it as 'neutral'.

Consistency. Participants generally considered that IF4SPLE does not alter the traditional role distribution among the SPL task force (question CON1). That said, in line with IF4SPLE, *pure::variant*'s employees seem to favor domain engineers over application engineers when conducting feedback specification and analysis (questions CON2 and CON4).

The next section looks at the feasibility of this approach, using *pure::variants* as the variability manager.

6. Proof-of-concept: implicit feedback in *pure::variants*

This section introduces *FEACKER* (a portmanteau for FEAture and traCKER), an extension for *pure::variants* along the IF4SPLE principles. *FEACKER* is publicly available at https://github.com/onekin/FEACKER.

The deployment diagram (i.e., the physical view Kim Hamilton, 2006) of the *FEACKER* system is presented in Fig. 10. The specification and transformation components are responsible for incorporating the tracking code, utilizing GA as the tracking framework. At run-time, variant execution on the *User PC* generates data in the GA log using the GA Javascript library, which is later analyzed by the analysis component of *FEACKER*. This component processes the feedback data and presents it through an attributed feature model, as described in Section 6.4.

The implementation of the solution relies on the extensibility options provided in the *pure::variants'* SDK. *Pure::variants* provides two different ways of extending its functionalities: the Java SDK[5] and the Javascript SDK.[6] The former allows you to extend the user interface using the standard mechanisms provided by the Eclipse platform and also offers a Java API that provides most of the non-user interface related functionality such as feature model validation. The latter is the one used by *FEACKER*, and it is focused on enabling developers to run complex operations on different *pure::variants* models.

FEACKER implements three extension points in *pure::variants*: specification, transformation, and analysis. It should be noted that *pure::var-*

[5] https://www.pure-systems.com/pv-update/additions/doc/latest/pv-pvesdk-manual.pdf

[6] https://www.pure-systems.com/pv-update/additions/doc/latest/pv-javascript-guide.pdf

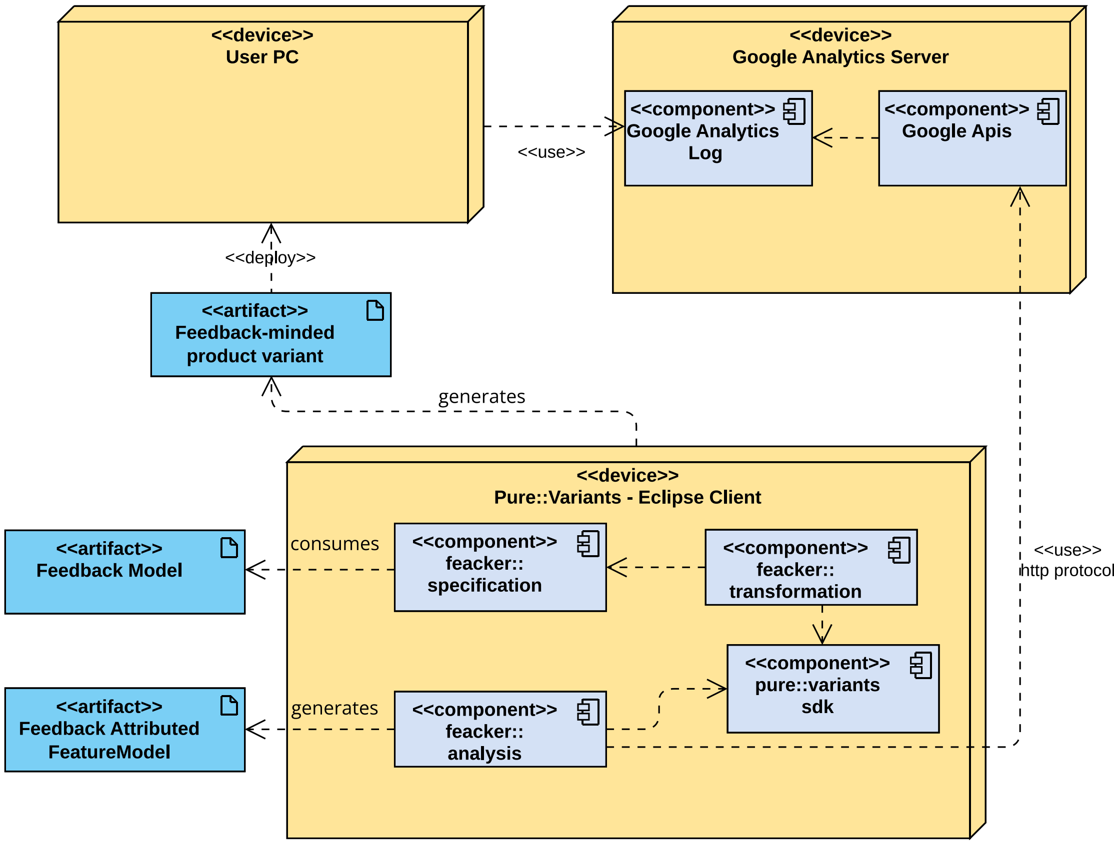

Fig. 10. *FEACKER*'s deployment diagram. *Pure::variants* is extended with three new components: *feacker::specification, feacker::analysis* and*feacker::transformation*. The specification and transformation components are responsible for integrating the tracking code, employing GA as the tracking framework. Subsequently, the derived feedback-minded variant is deployed. At runtime, when the variants are executed on the *User PC*, data is generated in the GA log. Afterward, this data undergoes analysis by the *FEACKER*'s analysis component.

iants uses pre-compilation directives for variability support and introduces the concept of a Family Model, which separates the SPL platform implementation through logical names referred to as 'model elements'. Details follow.

6.1. Feedback specification

FEACKER resorts to the YAML language for the concrete syntax of the Feedback Model introduced in Section 4.1. Fig. 11 provides a *myFirstfeedback.yaml* snippet for the running example. Clauses on lines 2–5 capture the Goal. Two properties have a documentation purpose (*audience* & *purpose*), whereas *target* and *context* have execution implications. As noted in Section 4, the target sets which features to track, whereas the context determines in which product variants this tracking should occur. In the example, the analysts are interested in tracking the usage of *Commenting* when it occurs in product variants that have *Highlighting* or *Emailing* (i.e., they include any of these features in their configuration models). Both properties hold a feature-based predicate similar to the one in pre-compilation directives (so far, only AND/OR are supported).[7]

Both predicates are consulted during product derivation to filter the *#ifdef* blocks subject to tracking as a result of matching the variability of interest (the target) in the appropriate variants (the context).

As for questions, they do not have any execution implications apart from structuring the metrics. The sample snippet introduced two questions: Q1 and Q2. The former is grounded on *Google-Scholar lookup ratio* metric. As for Q2, it is based on the counting of *doubleClickEnactment* ratio and *rightClickEnactment* ratio. The formula element defines how the metric is calculated using hit values. Note that the names of the hits are referenced using the '&' character.

Finally, hits are captured at various pointcuts, and subsequently, these pointcuts are identified through three clauses. *FileName* and *Path* single out the code file in *pure::variants*' Family Model. As for *the anchor*, it holds the line of code where the GA_INJECT directive is to be injected at transformation time.

6.2. Feedback transformation

In *pure::variants*, Product Derivation has two inputs: the configuration model (a .vdm file) and the SPL platform. The output of this process is a variant product, where *#ifdef* blocks that do not align with the configuration model are removed. This filtering of non-conforming blocks is achieved in *pure::variants* by selecting the 'File Processing' option. This is depicted on the right side of Fig. 12(b).

In *FEACKER*, Product Derivation extends *pure::variants'* through an additional step: Feedback Transformation. This transformation is triggered through the *FeedbackProcessing* option of *FEACKER*'s extension

[7] Context holds 'a contextual predicate' to delimit the variants whose configuration model should match this predicate. When Context is left empty, then the context of interest would be whole product line. *Target* holds 'an objectual predicate' to characterize the *#ifdef* blocks whose pre-compilation directives should satisfy this predicate. Thus, the slice of the platform code-base to be tracked is intensively defined by the *#ifdef* blocks whose pre-compilation directives satisfy the objectual predicate when deployed in variants matching the contextual predicate subject to the tracking (i.e., the slice of the platform code-base to be tracked). Back to the example, the tracking scope is that of *#ifdef* blocks whose pre-compilation directives hold *Commenting* when in variants that exhibit either *Highlighting* or *Emailing*.

```
goals:
  - audience: "Control Board"
    purpose: ["Scoping","Optimization"]
    target: Commenting
    context: Highligting or Emailing
    questions:
      - questionId: "Q1"
        description: "How is Commenting used?"
        metrics:
        - metricName: "Google-Scholar lookup  ratio"
          description: "What is the enactment ratio of Google-Scholar lookup  w.r.t. ..."
          formula: Total(&googleScholarLookupHit)/
            Total(&doubleClickEnactmentHit)+Total(&rightClickEnactmentHit)
      - questionId: "Q2"
        description: "How is Commenting enacted?"
        metrics:
        - metricName: "doubleClickEnactment ratio"
          description: "What is the enactment ratio of doubleClickEnactment w.r.t. ..."
          formula: Total(&doubleClickEnactmentHit)/
            Total(&doubleClickEnactmentHit)+Total(&rightClickEnactmentHit)
        - metricName: "rightClickEnactment"
          description: "What is the enactment ratio of rightClickEnactment w.r.t. ..."
          formula: Total(&rightClickEnactmentHit)/
            Total(&doubleClickEnactmentHit)+Total(&rightClickEnactmentHit)
hits:
    - name: "googleScholarLookupHit"
      pointCuts:
        - fileName: "CommentingForm.js"
          path: "app/scripts/annotationManagement/purposes/"
          anchor: |
           ...
    - name: "doubleClickEnactmentHit"
      pointCuts:
      - fileName: "ReadAnnotation.js"
        path: "app/scripts/annotationManagement/read/"
        anchor: |
          highlight.addEventListener('dbClick', () => {
            this.openCommentingForm(annotation)
            [*GA_INJECT*]
          })
      - fileName: "ReadAnnotation.js"
        path: "app/scripts/annotationManagement/read/"
        anchor: |
         {
          this.openCommentingForm(annotation)
          [*GA_INJECT*]
         }
     - name: "rightClickEnactmentHit"
       pointCuts:
        - fileName: "ReadComment.js"
          path: "app/scripts/annotationManagement/read/"
          anchor: |
            ...
```

Fig. 11. The Feedback Model for the goal: Analyze *Commenting* with the purpose of *scoping & optimization* with respect to *usage* from the point of view of the *Control Board* in the context of *variants that exhibit either Highlighting or Emailing*.

(see Fig. 12(d)). The output is also a variant, but now the variant code includes GA hits in accordance with the Feedback Model. Fig. 12(c) illustrates the progression of a code snippet. The process departs from the platform code-base. The Feedback Transformation injects the GA hits along with the Feedback Model. GA hits are composed of an interaction (*eventAction*) that happens during the usage of the feature (*eventCategory*) while using a given product (*variant*). Finally, the pre-compilation directives are removed from the code. The output is a feedback-minded variant. Refer to Appendix for the algorithm.

6.3. Feedback gathering

FEACKER resorts to GA for feedback gathering. Event data is collected into the GA feedback log. The utilization of GA presents both advantages and disadvantages. The benefits include GA's ability to comprehensively collect event data regarding visitors, traffic sources, content, and conversions. Additionally, GA provides advanced data visualization tools and it is highly scalable. On the downside, keeping event data at Google might jeopardize confidentiality.

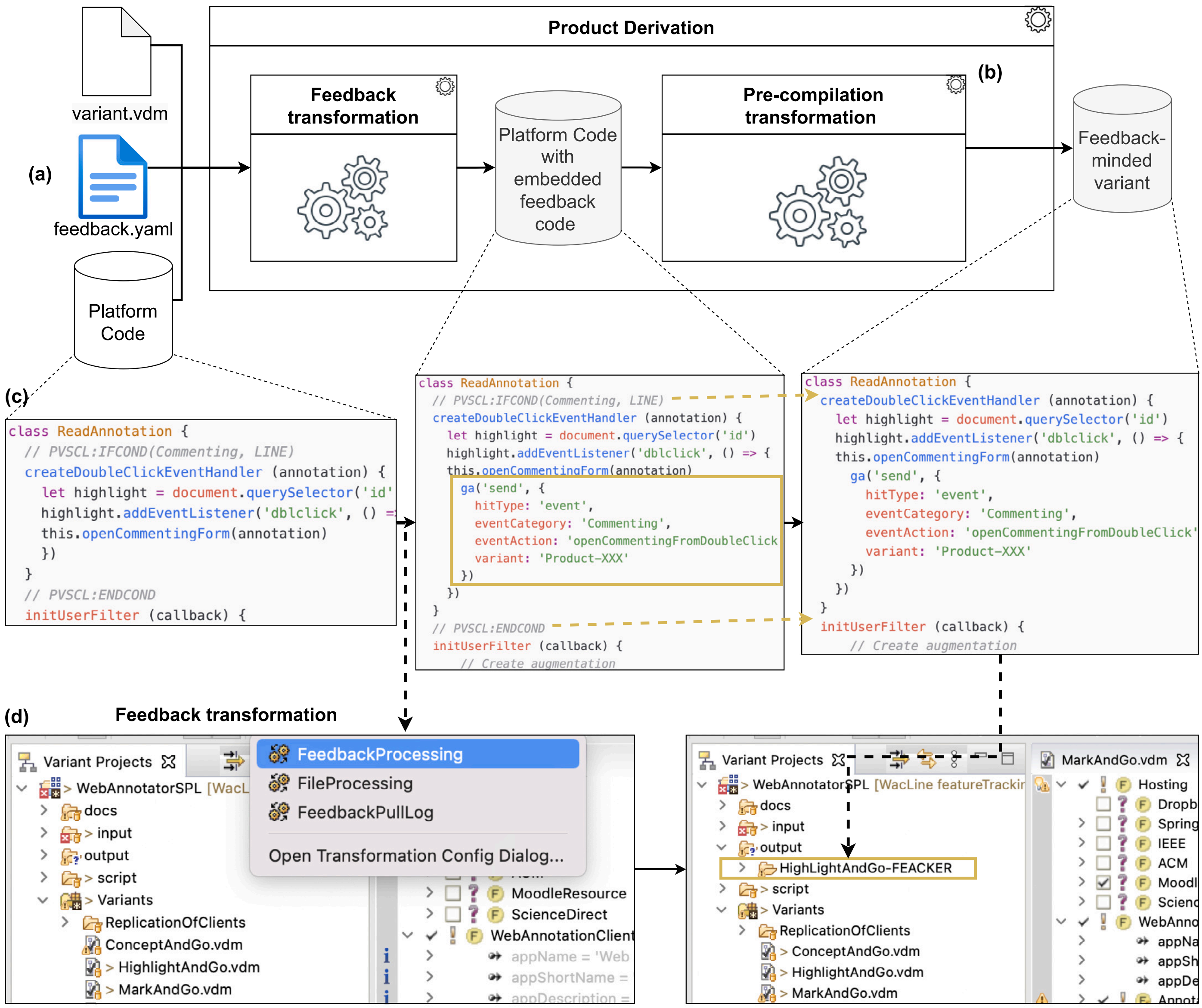

Fig. 12. *FEACKER*'s realization of Fig. 6. (a) The Feedback Transformation takes the platform code and a *yaml* file as input, and delivers a feedback-minded product variant. This involves two steps (b). First, the GA hits are injected into the to the corresponding *#ifdef* blocks. Second, the resulting code is next subjected to the traditional pre-compilation transformation that filters out *#ifdef* blocks that do not meet the configuration model (c). To this end, the *pure::variants* menu is extended with the *FeedbackProcessing* option (d). On selecting this option, *FEACKER* conducts the Feedback Transformation, and next invokes *pure::variants*' *FileProcessing* to launch the Pre-compilation Transformation. The result is a feedback-minded product variant, meaning that running this variant will generate usage hits to populate the feedback log.

6.4. Feedback analysis

GA provides data visualization tools to track single apps. By contrast, we pursue feature-centric dashboards that track an SPL portfolio. This is a topic in its own right. For the sake of completeness, this subsection explores the use of attributed feature models for feedback visualization.

In an attributed feature model, non-functional information is captured as attributes of features (Benavides et al., 2010). Examples include non-functional properties (e.g., cost, memory consumption, price) or rationales for feature inclusion. This permits leveraging the automated analysis of feature models by incorporating these usage parameters (Galindo et al., 2019). Likewise, we add 'usage' as a feature attribute. However, unlike other attributes such as cost or memory consumption, 'usage' is highly volatile and needs frequent updates. In this setting, databases resort to derived attributes, i.e., attributes that do not exist in the physical database but are derived at query time from the database. Regarding derived attributes, two steps are distinguished: definition (when the metric query is defined) and enactment (when the query is triggered).

FEACKER supports this vision, namely:

- definition time. First, analysts extend the feature model attributes for usage gathering. These attributes hold a GA query to obtain the aggregate for this metric using the GA's API. Fig. 13 (top) illustrates this point for our running example where the screenshot shows the *pure::variants* realization of the feature diagram at Fig. 2. *Commenting* is the target for three metrics: *DoubleClickEnactment ratio, RightClickEnactment ratio*, and *GoogleScholar lookup ratio*. Three namesake attributes are defined with their corresponding GA expressions that are equivalent to the formula defined in the Feedback Model.

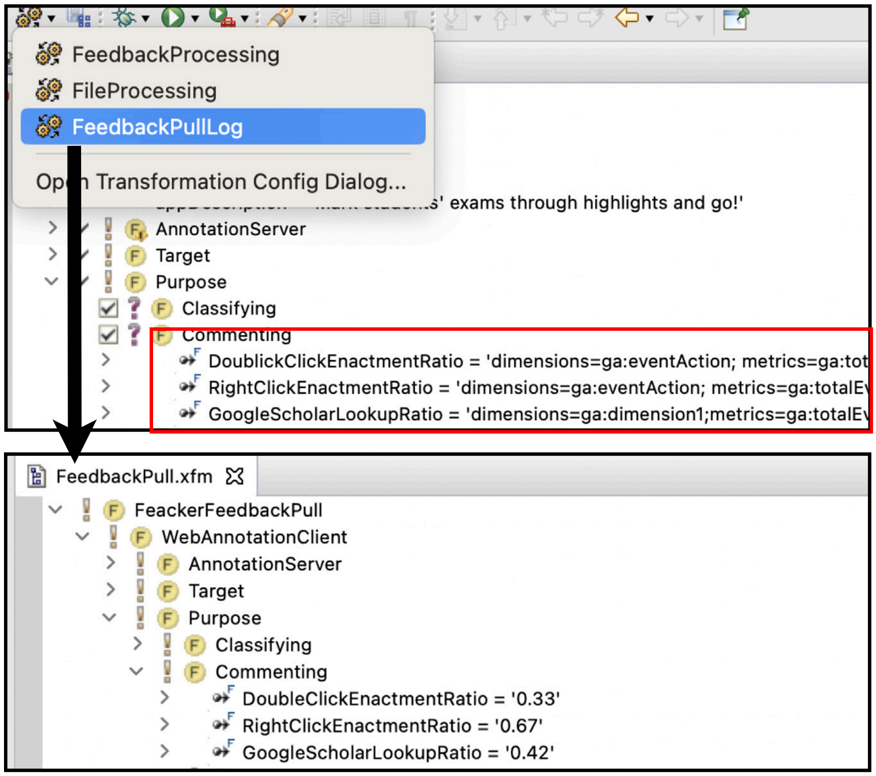

Fig. 13. Attributed Feature Models. At definition time, *Commenting* includes three attributes for collecting usage data. Each attribute holds the GA API calling parameters that calculate the metric value (top side). At enactment time (i.e., selecting *FeedbackPull* from the drop-down menu), the API calls are generated out of these parameters and enacted. Results are stored in *FeedbackPull.xfm*, a copy of the feature model where feature-usage attributes will show the metric values as returned by the API call (bottom side). FeedbackPull.xfm is updated each time the action FeedbackPull is selected.

- enactment time. It is up to the analysts to decide when to enact these queries. To this end, *FEACKER* adds a 'FeedbackLog-Pull' option to the *pure::variants* processing menu. When selected, *FEACKER* (1) retrieves the queries held as values of the attributes, (2) executes the queries, and (3) creates a clone of the feature model, where attributes now hold the metrics calculation results (noted as *FeedbackPull.xfm* in Fig. 13 (bottom)). These *xfm* files are held in the *FeedBackAnalysis* folder of the *pure::variant* installation. Refer to Appendix for the algorithm.

At present, query results are integrated into the model without further processing. Subsequent executions of 'FeedbackPull' will replace the existing clone with a new one that mirrors the current feedback log. In summary, using feature models falls short. First, it is not evident how to visualize metrics that involve multiple features (e.g., usage of F1 but not F2). Second, feature models offer a static view of the current variability, whereas feedback analysis necessitates a representation of how usage evolves. This calls for *pure::variants* to be enlarged with full-fledged dashboards. Alternatively, extending GA dashboards with feature-centric visualizations. This is still a work in progress.

7. Proof-of-value: Evaluation

7.1. Design of the evaluation

ADR emphasizes that authenticity is a more important ingredient than controlled settings (Sein et al., 2011). For an authentic evaluation, Sjøberg et al. (2002) introduce three factors: (1) realistic participants, (2) realistic tasks, and (3) a realistic environment. Sjøberg et al. recognize the difficulties of meeting all factors simultaneously, given the developing nature of the interventions in research.

FEACKER is not an exception; in fact, quite the opposite. However, the participation of practitioners from *WACline* and *pure-systems GmbH* provides a unique opportunity to assess the utility of *FEACKER*'s outcomes. The practitioners from *WACline* meet the first criteria of realistic participants, as *WACline* is web-based. They also used *FEACKER* in a real setting. However, *WACline*'s limited size may limit the results' generalizability. On the other hand, practitioners from *pure-systems GmbH* are well placed to assess the seamless integration of *FEACKER* with *pure::variants*. Furthermore, the company is a consultant for a broad range of customers, providing insights on the extent the approach could be generalized to settings other than Web-based variants. However, they have not used *FEACKER* in a real task. As a result, we opt for a hybrid evaluation that combines both quantitative and qualitative evaluation. Specifically (see Table 1):

Table 1
In the pursuit of a realistic evaluation.

	Realistic ...			
	Participants	Tasks	Environment	Instrument
pure-systems	✓	×	✓	TAM
WACline	✓	✓	×	Focus Group

- For quantitative evaluation, we resort to the Technology Acceptance Model (TAM) as the instrument (Davis et al., 2023) and *pure-systems GmbH* as the participant. TAM is founded upon the hypothesis that technology acceptance and use can be explained by a user's internal beliefs, attitudes, and intentions (Davis, 1989; Turner et al., 2010).
- For qualitative evaluation, we draw on Focus Groups as the instrument and *WACline* practitioners as the participants. Focus groups are suggested in Software Engineering as an empirical method for getting practitioner input and qualitative insights, particularly in the early phases of research for identifying issues (Kontio

et al., 2008). In this case, the evaluation resorts to a realistic task (i.e., the Web-based *Commenting* feature), and the focus groups are structured along the dissents expressed in the TAM evaluation.

This hybrid approach rests upon the assumption that both populations (i.e., *pure-systems GmbH* and *WACline*) share a common mindset that sustains that *the intention of use* of the former serves to guide *the real use* of the latter. This is undoubtedly a threat to validity, yet this likeness between both populations is (partially) supported by both using *pure::variants* as the variability manager.

7.2. TAM evaluation

Goal. The purpose of this study is to *assess* the perceived ease of use and usefulness of *FEACKER* with respect to *introducing IF4SPLE practices into pure::variants* from the point of view of *annotation-based SPL practitioners* in the context of *pure-systems GmbH*.

Participants. We sought the assistance of the same employees at *pure-systems GmbH* that participated in the evaluation of IF4SPLE described in Section 5. As they were involved in commercializing and developing *pure::variants*, we believed they were suitably positioned to assess its ease of use. Furthermore, half of the participants were consultants, exposing them to a wide range of domains, thereby providing a wider perspective on the *FEACKER*'s usefulness.

Design. *FEACKER* was evaluated as part of the same monthly seminar in which the IF4SPLE was previously evaluated (as described in Section 5). The seminar was structured as follows. An initial 90-minute session provided an overview of implicit feedback, the IF4SPLE process, and an in-depth examination of the *FEACKER's* components. A live demonstration of the tool followed this. Participants were encouraged to ask questions throughout the seminar. Finally, participants were allowed to interact with the tool and complete a TAM questionnaire.

Instrument. The study evaluates two internal variables of TAM: perceived usefulness and perceived ease of use.[8] Rationales follow. As consultants of companies of different industries, the participants were in a good position to provide an informed opinion on how much they believe using *FEACKER* would improve their job performance (perceived usefulness). As developers of *pure::variants*, the participants could provide an informed opinion on the degree of ease in using *FEACKER* for existing *pure::variants* users (perceived ease of use).

Results. Fig. 14 displays the chart for the variable perceived usefulness. Notice that usefulness is measured regarding the support given to the IF4SPLE process. The chart reveals a prevailing agreement among participants regarding the usefulness of *FEACKER*. However, the chart also indicates there is no full consensus on this matter (this will later serve to inform the Focus Group). As for perceived ease of use (see Fig. 15), results tend to be more conclusive on the seamlessness to which *FEACKER* is embedded within *pure::variants*. The results suggest that using *FEACKER* does not interfere with the existing GUI gestures of *pure::variants*.

Threats to validity.

- **Construct Validity** refers to the level of precision by which the variables specified in research assess the underlying constructs of concern. Here, the constructs are ease of use and usefulness. To mitigate its potential influence, the researchers utilized the TAM questionnaire. To ensure internal consistency, Cronbach's alpha is calculated for the questionnaire, which resulted in an α value of 0.62 and 0.97 for usefulness and ease of use, respectively. Cronbach alpha values of 0.7 or higher indicate acceptable internal consistency, with values above 0.9 perceived as redundancy of some questions.
- **Internal Validity** refers to how much the independent variable(s) or intervention was the actual cause of the changes observed in the dependent variable. Here, the intervention is *FEACKER*. Yet, other factors besides *FEACKER* might influence the results. If the participants are not randomly selected and their backgrounds are not representative of the target population, it may introduce selection bias. This bias can lead to an unrepresentative sample and limit the generalizability of the findings. In this regard, all the participants come from the same company, i.e., *pure-systems Gmbh*. We aim to reduce this threat in the future by being able to tap into practitioners coming from different companies. As for the background, it is reasonable to conjecture that their background is representative of the target population judging from the demographics in Fig. 7. Second, the clarity and understand ability of the questionnaire. To mitigate its potential influence, we resort to the TAM questionnaire, which has been widely employed and validated in assessing software engineering artifacts (Wallace and Sheetz, 2014).
- **External Validity** assesses to what extent the results obtained in this study can be generalized to other scopes different to the one approached in the study. We defer the discussion of this issue to Section 8.

7.3. Focus group evaluation

TAM measures *intention to use*, but it is not based on *the real use*, hence, contradicting the second principle of an authentic evaluation (Sjøberg et al., 2002). To mitigate this drawback, we resort to a second evaluation, now based on Focus Groups.

Goal. The purpose of this study is to *delve into* the divergent items of a previous TAM evaluation with respect to *introducing IF4SPLE practices into pure::variants* from the point of view of *annotation-based SPL practitioners* in the context of the *WACline* SPL.

Participants. The group was formed by three engineers with at least two-year experience in using *pure::variants*. All of them were part of *WACline*'s core-development team.

Design. Participants were initially introduced to the study objectives and the proposed intervention. Following this, they were asked to collaboratively create a Feedback Model. The model should approach analysis opportunities they were interested in. They came up with 18 different hits related to 11 features.[9] Feedback was gathered during a week about *WACline* products running in a sandbox framework. So collected feedback was presented through the usage-based attributed feature model shown in Fig. 13. Once the usage feedback was collected, the focus groups started. During the session, one of the authors conducted the focus group while another author took notes and audio-recorded the discussion.

Instrument. The focus group was structured along those issues that rose the most significant divergence in the TAM evaluation (see Fig. 14 and Fig. 15). Each issue was turned into a question for the focus group, namely:

- **Is the Feedback Model able to capture your feedback needs effectively?** This question accounts for the TAM's US1 statement: *Using FEACKER would enable me to accomplish IF4SPLE tasks more quickly*. Fig. 14 depicts the divergences in the answers: disagree (1), neutral (3), agree (3), and strongly disagree (1). Among the tasks introduced in the IF4SPLE, the more labor-intensive one is feedback specification since it requires knowledge about the

[8] A replication package with the questionnaire and the response data is available: https://doi.org/10.5281/zenodo.8187116

[9] The resultant Feedback Model can be found in a branch of *WACline*'s repository: https://github.com/onekin/WACline/tree/featureTracking/WACline

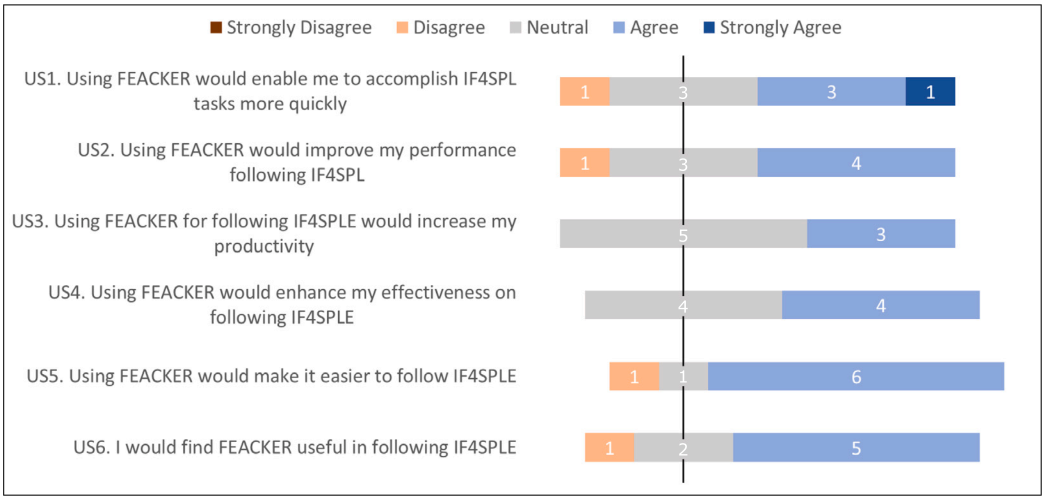

Fig. 14. *FEACKER*'s perceived usefulness.

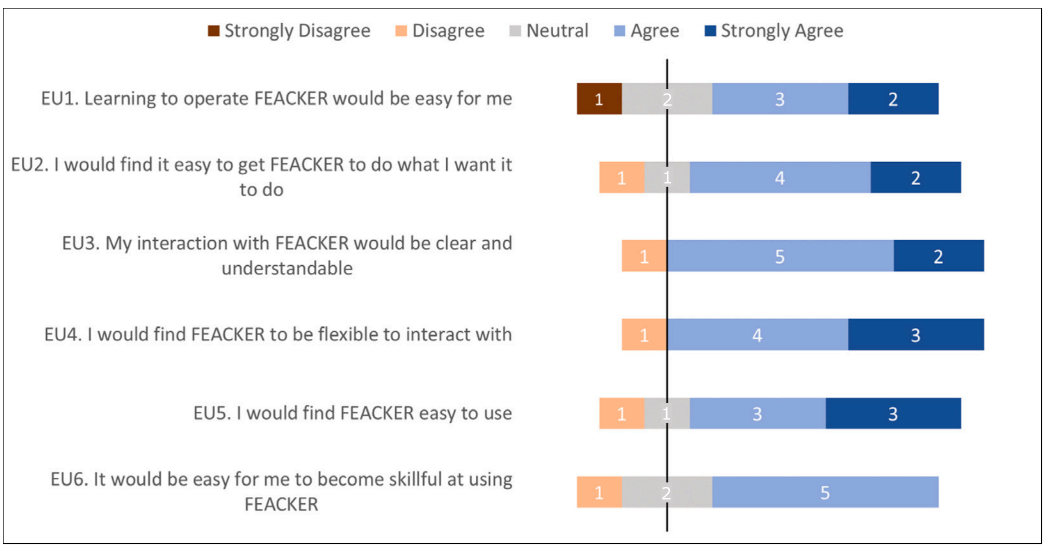

Fig. 15. *FEACKER*'s perceived ease of use.

feature model, the SPL product portfolio, and the code of the feature(s) at hand. Hence, the focus question delves into the extent *FEACKER*'s Feedback Model can directly capture the needs of the analysts.

- **How seamlessly were IF4SPLE's new tasks integrated with your SPLE practices?** This question accounts for the TAM's US2 statement: *Using FEACKER would improve my performance following IF4SPLE*. Fig. 14 depicts the divergences in the answers: disagree (1), neutral (3), and agree (4). IF4SPLE proposes the need for an analysis space and the consequent tasks in SPLE. The way that these tasks are integrated into the traditional workflow may have a direct influence on the performance of the team. Hence, the focus question seeks to explore the impact of IF4SPLE tasks on the performance of domain engineers.
- **How seamless was *FEACKER* integrated with *pure::variants*' gestures?** This question accounts for the TAM's EU1 statement: *Learning to operate FEACKER would be easy for me*. Fig. 14 depicts the divergences in the answers: strongly disagree (1), neutral (2), agree (3), and strongly agree (2). Being *FEACKER* an extension can disrupt the way of interacting with *pure::variants*. Thus, this question tries to determine how much *FEACKER* disturbs existing *pure::variants*' gestures.

Results.

- **Is the Feedback Model able to capture your feedback needs effectively?** Participants indicated that most of their feedback expressions refer to a single feature. No cases popped up with AND/OR expressions. Two participants missed the ability to record values of variables at execution time. This would allow capturing the number of items a user has at the same time to operate within the GUI or capturing values that could make the system fail.
- **How seamlessly were IF4SPLE's new tasks integrated with your SPLE practices?** Participants agreed on the feedback functionality being subject to the same reuse principles as domain functionality and hence, being moved to DE. As one participant put it: "If testing is mainly a DE activity, why should feedback analysis differ? After all, usage feedback informs feature evolution, sort of perfective maintenance". That said, they expressed two worries. First, the Feedback Model is fragile upon changes in the *#ifdef* blocks. This introduces a dependency between code developers and feedback analysts. How severe is this limitation? Participants observe that feedback analysis tends to be conducted for mature features where customer usage drives feature evolution rather than feature 'upbringing'. For mature features, the

expectation is that code would be relatively stable and hence, with a low risk of interference between feedback analysis and feature developers. Second and most importantly, upgrades to the Feedback Model would only take effect by deriving the products again. This might be a severe concern for some installations, but, after all, this is what continuous deployment is supposed to be. Finally, some exciting issues arose with no clear answer: How is the evolution of Feedback Models conducted? How would different Feedback Models co-exist? Should variability be engineered within the Feedback Model itself? What strategy would be to face large feature models with intensive feedback needs? Should techniques similar to those for SPL testing be used?
- **How seamlessly was *FEACKER* integrated with *pure::variants*' gestures?** Participants mostly appreciated conducting feedback analysis without leaving *pure::variants*. This seems to suggest that variability managers should not overlook this concern.

Nevertheless, some issues emerged:

- Feedback Model. All participants found it cumbersome to specify. Difficulties came from Feedback Models' dependency on the *#ifdef* blocks, specifically, the anchor specification. This limitation might be eventually mitigated through dedicated editors.
- Feedback Transformation. Participants found the approach intuitive and akin to the *modus operandi* of *pure::variants*.
- Attributed Feature Models. Participants foresaw the benefits of introducing feature usage for more sophisticated feature analysis. However, feature models are very poor as dashboards. Participants indicate the need to enhance variability managers with this functionality[10] or, instead, tap into existing dashboard applications through dedicated connectors.

7.4. Threats to validity

We follow here the threats to validity for Qualitative Research as described by Maxwell (1992):

- **Descriptive Validity** refers to the accuracy and completeness of the data. To mitigate this threat, we took notes about participants' reactions and recorded the session's audio to facilitate further analysis. Moreover, we asked for clarifications from the participants during the discussion. A major concern is the small number of participants (n=3), which is below the recommended value of 8 participants (Kontio et al., 2008). Additionally, there is a significant threat to internal and external validity as all the members are from the same company. This is partially due to the pursuit of realism and purposive sampling. Specifically, we resort to Expert Sampling where participants are considered experts in both *pure::variants* and *Google Analytics*, and not only SPLs. Favoring realism over quantity is supported by the literature on focus groups, which recommends purposive sampling as the method for participant recruitment (Morgan, 1997).
- **Interpretive Validity** refers to how well researchers capture participants' meaning of events or behaviors without imposing their perspective. It should be noted that the problem was pointed out by the practitioners themselves when attempting to accord *pure::variants* and GA. To improve interpretative validity, we began the session with a brief presentation on the study's objectives and the proposed intervention. This presentation was aimed at establishing common terminology and avoiding misunderstandings.
- **Reproducibility**. *FEACKER* and *WACline* are readily accessible to the public through download. The additional infrastructure utilized in this study, including *pure::variants* and GA, is widely used among practitioners, facilitating the dissemination of this work's results.

[10] It rests to be seen whether *pure::variants*' tabular view of features which includes attributes, might provide a better fit.

Table 2
Contextual characterization of the local experience in WACline.

Technical environment	Programming language	*JavaScript*
	Branching strategy	Grow-and-prune
	Variability manager	*pure::variants*
	Tracker Manager	*Google Analytics*
SPL Attributes	Lifespan	5 years
	Size (approx.)	86 features & 8 products
	Domain	Web
	Variability mechanism	Annotation-based
Implicit Feedback	Purpose	Scoping

8. Generalization

Following ADR, the situated learning from the research project should be further developed into general solution concepts for a class of field problems (Sein et al., 2011). Sein et al. suggest three levels for this conceptual move:

- generalization of the problem instance, i.e., to what extent is *implicit feedback* a *problem* for SPL other than *WACline*?
- generalization of the solution instance, i.e., to what extent is *FEACKER* a *solution* to implicit feedback in SPLs?
- derivation of design principles, i.e., what sort of design knowledge can be distilled from the *FEACKER* experience that might inform variability managers other than *pure::variants*.

The rest of this section tackles these questions.

8.1. Generalization of the problem instance

Generalizing from a local experience starts by identifying the contextual parameters that might have influenced the solution and its utility. Table 2 gathers what we believe are the main contextual parameters. We also refer to the threats to validity that arose during the Focus Group. Our solution applies as far as the participants were knowledgeable about *pure-systems GmbH* and *Google Analytics*. The question arises as to whether our setting is unique or if other SPL installations can share it.

The first consideration is the domain: the Web. The Web is certainly a pioneering domain in applying continuous deployment using implicit feedback, such as in mobile app development (Xiao et al., 2021), online training platform (Astegher et al., 2021), or mobile and web contexts (Oriol et al., 2018). However, we argue that the interest in implicit feedback is not limited to the Web. Clements' popular definition of an SPL as addressing a particular market segment implies that the evolution of this market segment should naturally go hand in hand with the evolution of the SPL (Clements and Northrop, 2002). Therefore, SPL scoping, which involves deciding on the features to be covered by the SPL, must be continuous to keep pace with this market segment. If SPL scoping is continuous, then implicit feedback will become a primary informant of how this market segment evolves. And if so, it is likely that the problem of implicit feedback found in *WACline* is shared by other installations.

It is worth noting that *WACline*'s incentive for implicit feedback is to inform scoping. However, usage data is also useful for troubleshooting, fault prediction, supporting network operations or development activities (Dakkak et al., 2021b, 2022b), and enabling continuous experimentation (Mattos et al., 2018). Collected data provides valuable insights into the real-world behavior and performance of embedded systems, allowing for the identification of potential issues and the development of solutions to improve performance. Additionally, models and simulations used in the evolution of embedded systems can be validated and refined, providing a more robust and accurate understanding of the system's behavior (Dakkak et al., 2021b, 2022b). We can then conjecture that implicit feedback would also be of interest to embedded systems.

Table 3
Design principles for implicit feedback in variability managers.

ID	Provide *variability managers* with...	For domain engineers to...	Realization in *pure::variants*
DP1	... a goal-centric feedback specification model	... structure feedback directives using GQM	a *yaml* realization of GQM constructs
DP2	... a feature-based feedback model	... align feedback with other SPLE tasks	yaml's *target* and *context* clauses in terms of features
DP3	... feedback transformations	... account for 'separation of concerns' to avoid polluting the platform code	A two-step Product Derivation
DP4	... feature-based feedback dashboards	... track SPL-wide metrics	Feature models with derived attributes

8.2. Generalization of the solution instance

FEACKER is an intervention for *pure::variants* as the variability manager, GA as the event tracker, Web-based product portfolios, and integrated product deployment. The following paragraphs discuss the limitations of such characterization in the generalizability of *FEACKER*.

Technological limitations. The focus on *pure::variants* limits our solution to annotative SPLs. Other approaches using feature-oriented programming or component-based SPLs would need to resort to other means as our approach is heavily based on pre-compilation directives. As for GA, the *FEACKER*'s Feedback Model in Fig. 5 might be biased by hit specification in GA. Other trackers like Matomo or Zoho might have different expressiveness, which would involve changes in the Feedback Model.

Limitations of organization. *FEACKER* is designed for organizations that have access to SPL variants once they are deployed. However, in order to collect usage data, a data pipeline must be established that connects to the deployed variants within the customer's network. These variants may be located in a secure intranet with multiple security measures to prevent unauthorized access (Dakkak et al., 2021b). The setup for each customer will be unique and require customized configurations to connect the variants to the data collectors and the pipeline. This raises the issue of data completeness, i.e., how well the collected data represents the full range of products used by customers. If access to variant usage is limited and only a subset of variants can be sampled, data completeness might be at risk. The challenge in these cases is to gather data from a representative sample of variants that accurately reflects the SPL platform.

Domain limitations. *FEACKER* is an intervention for Web-based product portfolios. Using *FEACKER* in embedded and cyber–physical domains introduces additional challenges:

- *Performance impact.* The limited resources of embedded and cyber–physical systems make it crucial to optimize data collection, as these activities can consume internal resources. One solution is to collect data during low-traffic hours. The situation can be exacerbated in SPLs, where each variant configuration may suffer different performance impacts, calling for a bespoke solution to minimize the performance impact on each deployed variant (Dakkak et al., 2022a; Mattos et al., 2018).
- *Data dependability.* Embedded systems generate a wide range of data types, including sensor readings, network status, performance metrics, and behavioral patterns. Assessing the quality of just one data type might be insufficient. A thorough analysis of data quality necessitates the examination of interrelationships and correlations among multiple data sources. The challenge lies in the fact that each product configuration is equipped with different sensors and data sources, which calls for the feedback collection process to be cognizant of the distinct settings and requirements of each configuration (Dakkak et al., 2021b).

In summary, this work sought a global perspective in introducing feedback analysis in SPLs. Each of the feedback tasks (i.e., specification, transformation, gathering, and analysis) is a subject in itself, raising diverse questions:

- How could feature usage leverage existing feature-analysis techniques? Could dependency between two features be weighted based on the (co)usage of these two features? Could this co-usage between features be used by configuration assistants to recommend co-features?
- How could SPL evolution become 'continuous' by tracking feature usage? Which sort of metrics can be most helpful in tracking SPL evolution?
- How can existing experiences on analysis dashboards be tuned for platform-based feedback?
- To what extent does implicit feedback account for the same benefits as in product-based development?

8.3. Derivation of design principles

Design principles should reflect knowledge of both IT and human behavior (Gregor et al., 2020). Consequently, a design principle should provide cues about the effect, such as allowing for implicit feedback at the platform level, the cause, such as feedback transformation, and the context where this cause can be expected to yield the effect for the target audience, such as domain engineers. Table 3 outlines the main design principles. The following are considered as main design decisions: structuring the feature model along the GQM model, characterizing the Feedback Model in terms of features, supporting the Feedback Model as a full-fledged model that can be executed by transformation, and providing feature-based feedback dashboards.

9. Conclusion

We explored extending feedback practices to the SPL platform by examining how these practices might impact both the SPLE processes and variability manager tools. As for how platform-based feedback can be integrated into SPLE development, we introduced IF4SPLE by placing features at the core of the feedback analysis and resorting to a derivation approach for injecting the trackers into the variants codebase (i.e., platform-based feedback). Next, we studied how variability managers can support platform-based feedback. To this end, we developed *FEACKER*, a proof-of-concept extension to *pure::variants*, which is publicly available for inspection and use. The evaluation involved both real practitioners in a TAM evaluation (n=8) and a focus group (n=3). The results suggest that the approach aligns seamlessly with current practices but raises several issues regarding its generalizability.

The next steps are driven by this limitation in generalizability. Firstly, the current approach heavily relies on pre-compilation directives, which restricts the solution to annotative SPLs, potentially disregarding feature-oriented programming or component-based SPLs. Additionally, the domain limitations of *FEACKER*, designed for web-based product portfolios, may pose challenges when applied to embedded and cyber–physical domains. Ultimately, these efforts aim to spark

Algorithm 1: Feedback Transformation takes a feedback model, the platform code and a variant configuration model as an input and returns the platform code with embedded feedback hits.

```
function PlatformCode feedbackTransformation( FeedbackModel feedbackModel, PlatformCode platformCode,
     VariantConfigurationModel variantConfigurationModel){
    //Extract selected and deselected features from the variant configuration
    Map<Feature,Boolean> selectedFeatures = extractFeatures(variantConfigurationModel)
    //For each goal check whether the input variant is part of the goal context and if it satisfies the target
         expression
    foreach(goal in feedbackModel.goals){
        if(isExpressionTrue(feedbackModel.context, selectedFeatures) and isExpressionTrue(feedbackModel.
             target, selectedFeatures)){
            //Create the directory for the platform code with the embedded GA code
            Platform feedbackPlatformCode = copyDirectory(platformCode)
                //For each question in the goal, get its metrics
                foreach (question in goal.questions){
                    foreach (metric in question.metrics){
                        //For each hit find where the GA code should be injected
                        foreach (hit in metric.hits){
                            foreach (pointcut in metric.pointCuts){
                                File file = findFile(feedbackPlatformCode, pointcut.fileName)
                                foreach (ifdefBlock in file){
                                    //Check if the #ifdef block satisfies the target
                                    if(doesIfdefMatchExpression(goal.target, ifdefBlock)){
                                            //Generate the google analytics hit code and add it to the given file
                                            Integer anchorLine = findAnchor(pointcut.anchor, ifdefBlock, file)
                                            addGoogleAnalyticsHit(anchorLine, hit, file)
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    return feedbackPlatformCode
}
```

discussions among researchers and practitioners about the challenges of integrating implicit feedback in SPLs. We believe that fostering such debates can pave the way for bringing continuous deployment into SPLE.

CRediT authorship contribution statement

Oscar Díaz: Conceptualization, Funding acquisition, Methodology, Project administration, Resources, Supervision, Validation, Writing – original draft, Investigation, Writing – review & editing. **Raul Medeiros:** Conceptualization, Data curation, Validation, Resources, Software, Investigation, Writing – review & editing, Visualization. **Mustafa Al-Hajjaji:** Resources, Validation, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

We have publicly published the artifacts in GitHub and the data on Zenodo: https://github.com/onekin/FEACKER and https://doi.org/10.5281/zenodo.8187116.

Acknowledgments

We would like to extend our gratitude to both WACline's and *pure-systems*' employees for their invaluable contributions to this research. We are grateful for their time and effort. This work is co-supported by MCIN/AEI/10.13039/501100011033 and the "European Union NextGenerationEU/PRTR" under contract PID2021-125438OB-I00. Raul Medeiros enjoys a doctoral grant from the Ministry of Science and Innovation of Spain - PRE2019-087324.

Appendix. Algorithms

Feedback Transformation (Algorithm 1): The process starts by creating a copy of the platform code. Then, it filters the goals specified in the Feedback Model (*yaml* file) based on the target and context that match the configuration model. The corresponding hits specifications are processed for the goals that pass this filter. The hit pointcuts are determined using the file names and anchors defined in the Feedback Model, with the file name corresponding to an artifact of the Family Model and the anchor serving to single out the code fragment where the GA hit is to be injected. Finally, *FEACKER* generates the GA hits and inserts them into the appropriate code lines. *Feedback Analysis (Algorithm 2):* The algorithm gets as input a feature model. First, the analysis component searches for GA query parameters in the given feature model. The value of these attributes (i.e., the GA API query) is then used as the input to call the GA API. Finally, a copy of the feature model is created and the raw results obtained from GA API replace the query parameter value of the attributes in the cloned feature model.

Algorithm 2: Feedback Analysis, takes a feature model with GA query parameters on it and returns a cloned feature model with the raw results of those GA query.

```
function FeatureModel feedbackPull(FeatureModel implicitFeatureModel){
    //Search for Google Analytics usage attributes in the implicit feature model.
    Map<String, String> usageAttributeMap = findFeedbackQueryParameters(implicitFeatureModel)
    Map<String, String> feedackResultsAttributeMap = new Map()
    //For each attribute, get its value, call GA API and store the result
    for each (usageAttribute in usageAttributeMap){
        String googleAnalyticsQuery = usageAttribute.value
        String feedbackData = googleAnalyticsApi.getFeedbackData(googleAnalyticsQuery)
        //Store GA API results in an auxiliary map
        feedackResultsAttributeMap[usageAttribute.name]=feedbackData
    }
    //Create the cloned feature model
    FeatureModel clonedFeatureModel=copyFeatureModel(implicitFeatureModel)
    //For each attribute replace the value with the result obtained from GA API
    for each (feedbackResultAttribute in feedackResultsAttributeMap){
        featureModelAttribute = findFeedbackAttributeInFeatureModel(feedbackResultAttribute.name,
             usageAttributedFeatureModel)
        featureModelAttribute.setValue(feedbackResultAttribute.value)
    }
    //Save the changes of the cloned feature model
    usageAttributedFeatureModel.saveChanges()
    return clonedFeatureModel
}
```

References

Apel, S., Batory, D.S., Kästner, C., Saake, G., 2013. Feature-Oriented Software Product Lines - Concepts and Implementation. Springer, http://dx.doi.org/10.1007/978-3-642-37521-7.

Astegher, M., Busetta, P., Perini, A., Susi, A., 2021. Specifying requirements for data collection and analysis in data-driven RE. A research preview. In: Dalpiaz, F., Spoletini, P. (Eds.), Requirements Engineering: Foundation for Software Quality - 27th International Working Conference, REFSQ 2021, Essen, Germany, April (2021) 12-15, Proceedings. Springer, pp. 182–188. http://dx.doi.org/10.1007/978-3-030-73128-1_13.

Basili, V.R., Rombach, H.D., 1988. The tame project: Towards improvement-oriented software environments. IEEE Trans. Softw. Eng. 14, 758–773.

Benavides, D., Segura, S., Cortés, A.R., 2010. Automated analysis of feature models 20 years later: A literature review. Inf. Syst. 35, 615–636. http://dx.doi.org/10.1016/j.is.2010.01.001.

Beuche, D., 2019. Industrial variant management with pure::variants. In: ACM International Conference Proceeding Series. Association for Computing Machinery, New York, New York, USA, pp. 1–3. http://dx.doi.org/10.1145/3307630.3342391.

Caria, F., Mathiak, B., 2020. Annotation in Digital Humanities. Springer International Publishing, Cham, pp. 39–50. http://dx.doi.org/10.1007/978-3-030-15200-0_3.

Clements, P., Northrop, L.M., 2002. Software Product Lines - Practices and Patterns. In: SEI series in software engineering, Addison-Wesley.

Dakkak, A., Bosch, J., Olsson, H.H., 2022a. Controlled continuous deployment: A case study from the telecommunications domain. In: ICSSP/ICGSE 2022: 16th International Conference on Software and System Processes and 17th International Conference on Global Software Engineering. Virtual Event / Pittsburgh, PA, USA, May (2022) 19-20, ACM, pp. 24–33. http://dx.doi.org/10.1145/3529320.3529323.

Dakkak, A., Mattos, D.I., Bosch, J., 2021a. Perceived benefits of continuous deployment in software-intensive embedded systems. In: IEEE 45th Annual Computers, Software, and Applications Conference. COMPSAC 2021, Madrid, Spain, July (2021) 12-16, IEEE, pp. 934–941. http://dx.doi.org/10.1109/COMPSAC51774.2021.00126.

Dakkak, A., Munappy, A.R., Bosch, J., Olsson, H.H., 2022b. Customer support in the era of continuous deployment: A software-intensive embedded systems case study. In: 46th IEEE Annual Computers, Software, and Applications Conferenc. COMPSAC 2022, Los Alamitos, CA, USA, June 27 - July 1 2022, IEEE, pp. 914–923. http://dx.doi.org/10.1109/COMPSAC54236.2022.00143.

Dakkak, A., Zhang, H., Mattos, D.I., Bosch, J., Olsson, H.H., 2021b. Towards continuous data collection from in-service products: Exploring the relation between data dimensions and collection challenges. In: 28th Asia-Pacific Software Engineering Conference. APSEC 2021, Taipei, Taiwan, December (2021) 6-9, IEEE, pp. 243–252. http://dx.doi.org/10.1109/APSEC53868.2021.00032.

Davis, F.D., 1989. Perceived usefulness, perceived ease of use, and user acceptance of information technology. MIS Q. 13, 319–340, URL: http://misq.org/perceived-usefulness-perceived-ease-of-use-and-user-acceptance-of-information-technology.html.

Davis, F.D., Granić, A., Marangunić, N., 2023. The technology acceptance model 30 years of tam. Technology.

Díaz, O., Montalvillo, L., Medeiros, R., Azanza, M., Fogdal, T., 2022. Visualizing the customization endeavor in product-based-evolving software product lines: A case of action design research. Empir. Softw. Eng. 27 (75).

Fitzgerald, B., Stol, K., 2017. Continuous software engineering: A roadmap and agenda. J. Syst. Softw. 123, 176–189. http://dx.doi.org/10.1016/j.jss.2015.06.063.

Galindo, J.A., Benavides, D., Trinidad, P., Gutiérrez-Fernández, A.M., Ruiz-Cortés, A., 2019. Automated analysis of feature models: Quo vadis? Computing 101, 387–433.

Google, 2016. Google analytics | hit. URL: https://support.google.com/analytics/answer/6086082?hl=en.

Gregor, S., Kruse, L.C., Seidel, S., 2020. Research perspectives: The anatomy of a design principle. J. Assoc. Inf. Syst. 21 (2), URL: https://aisel.aisnet.org/jais/vol21/iss6/2.

Hellebrand, R., Schulze, M., Ryssel, U., 2017. Reverse engineering challenges of the feedback scenario in co-evolving product lines. In: Proceedings of the 21st International Systems and Software Product Line Conference. SPLC 2017, Volume B, Sevilla, Spain, September (2017) 25-29, ACM, pp. 53–56. http://dx.doi.org/10.1145/3109729.3109735.

Hoffmann, P., Spohrer, K., Heinzl, A., 2020. Analyzing Usage Data in Enterprise Cloud Software: An Action Design Research Approach. Springer International Publishing, Cham, pp. 257–274. http://dx.doi.org/10.1007/978-3-030-45819-5_11.

Johanssen, J.O., Kleebaum, A., Bruegge, B., Paech, B., 2018. Feature crumbs: Adapting usage monitoring to continuous software engineering. In: Product-Focused Software Process Improvement - 19th International Conference, PROFES 2018, Wolfsburg, Germany, November (2018) 28-30, Proceedings. Springer, pp. 263–271. http://dx.doi.org/10.1007/978-3-030-03673-7_19.

Johanssen, J.O., Kleebaum, A., Bruegge, B., Paech, B., 2019. How do practitioners capture and utilize user feedback during continuous software engineering? In: 27th IEEE International Requirements Engineering Conference. RE 2019, Jeju Island, Korea (South), September (2019) 23-27, IEEE, pp. 153–164. http://dx.doi.org/10.1109/RE.2019.00026.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J., Irwin, J., 1997. Aspect-oriented programming. In: Aksit, M., Matsuoka, S. (Eds.), ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June (1997) 9-13, Proceedings. Springer, pp. 220–242. http://dx.doi.org/10.1007/BFb0053381.

Kim Hamilton, R.M., 2006. Learning Uml 2.0. O'Reilly, Pub Date: April.

Kontio, J., Bragge, J., Lehtola, L., 2008. The Focus Group Method As an Empirical Tool in Software Engineering. Springer, pp. 93–116. http://dx.doi.org/10.1007/978-1-84800-044-5_4.

Krueger, C.W., Clements, P., 2018. Feature-based systems and software product line engineering with gears from biglever. In: Proceeedings of the 22nd International Systems and Software Product Line Conference - Volume 2. SPLC 2018, Gothenburg, Sweden, September (2018) 10-14, ACM, pp. 1–4. http://dx.doi.org/10.1145/3236405.3236409.

Liang, W., Qian, W., Wu, Y., Peng, X., Zhao, W., 2015. Mining context-aware user requirements from crowd contributed mobile data. In: Proceedings of the 7th Asia-Pacific Symposium on Internetware. Internetware 2015, Wuhan, China, November 6 2015, ACM, pp. 132–140. http://dx.doi.org/10.1145/2875913.2875933.

Lindohf, R., Krüger, J., Herzog, E., Berger, T., 2021. Software product-line evaluation in the large. Empir. Softw. Eng. 26, 30. http://dx.doi.org/10.1007/s10664-020-09913-9.

López, L., Manzano, M., Gómez, C., Oriol, M., Farré, C., Franch, X., Martínez-Fernández, S., Vollmer, A.M., 2021. Qasd: A quality-aware strategic dashboard for supporting decision makers in agile software development. Sci. Comput. Program. 202, 102568. http://dx.doi.org/10.1016/j.scico.2020.102568.

Maalej, W., Happel, H., Rashid, A., 2009. When users become collaborators: towards continuous and context-aware user input. In: Arora, S., Leavens, G.T. (Eds.), Companion To the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2009, October (2009) 25-29, Orlando, Florida, USA, ACM, pp. 981–990. http://dx.doi.org/10.1145/1639950.1640068.

Martínez-Fernández, S., Vollmer, A.M., Jedlitschka, A., Franch, X., López, L., Ram, P., Rodríguez, P., Aaramaa, S., Bagnato, A., Choras, M., Partanen, J., 2019. Continuously assessing and improving software quality with software analytics tools: A case study. IEEE Access 7, 68219–68239. http://dx.doi.org/10.1109/ACCESS.2019.2917403.

Mattos, D.I., Bosch, J., Olsson, H.H., 2018. Challenges and strategies for undertaking continuous experimentation to embedded systems: Industry and research perspectives. In: Agile Processes in Software Engineering and Extreme Programming - 19th International Conference, XP 2018, Porto, Portugal, May (2018) 21-25, Proceedings. Springer, pp. 277–292. http://dx.doi.org/10.1007/978-3-319-91602-6_20.

Maxwell, J., 1992. Understanding and validity in qualitative research. Harv. Educ. Rev. 62, 279–301.

Medina, H., Díaz, O., Garmendia, X., 2022. Wacline: A software product line to harness heterogeneity in web annotation. SoftwareX 18, 101090. http://dx.doi.org/10.1016/j.softx.2022.101090.

Morgan, D.L., 1997. Focus Groups As Qualitative Research. Qualitative Research Methods. SAGE Publications, Thousand Oaks, CA, http://dx.doi.org/10.4135/9781412984287.

Olsson, H.H., Bosch, J., 2013. Towards data-driven product development: A multiple case study on post-deployment data usage in software-intensive embedded systems. In: Fitzgerald, B., Conboy, K., Power, K., Valerdi, R., Morgan, L., Stol, K. (Eds.), Lean Enterprise Software and Systems - 4th International Conference, LESS 2013, Galway, Ireland, December (2013) 1-4, Proceedings. Springer, pp. 152–164. http://dx.doi.org/10.1007/978-3-642-44930-7_10.

Oriol, M., Stade, M.J.C., Fotrousi, F., Nadal, S., Varga, J., Seyff, N., Abelló, A., Franch, X., Marco, J., Schmidt, O., 2018. FAME: supporting continuous requirements elicitation by combining user feedback and monitoring. In: 26th IEEE International Requirements Engineering Conference. RE 2018, Banff, AB, Canada, August (2018) 20-24, IEEE Computer Society, pp. 217–227. http://dx.doi.org/10.1109/RE.2018.00030.

Plaza, B., 2011. Google analytics for measuring website performance. Tour. Manag. 32, 477–481.

Pohl, K., Böckle, G., van der Linden, F., 2005. Software Product Line Engineering - Foundations, Principles, and Techniques. Springer, http://dx.doi.org/10.1007/3-540-28901-1.

Redman, T.C., 2013. Data quality management past, present, and future: Towards a management system for data. In: Sadiq, S.W. (Ed.), Handbook of Data Quality, Research and Practice. Springer, pp. 15–40. http://dx.doi.org/10.1007/978-3-642-36257-6_2.

Rehm, H.L., Berg, J.S., Brooks, L.D., Bustamante, C.D., Evans, J.P., Landrum, M.J., Ledbetter, D.H., Maglott, D.R., Martin, C.L., Nussbaum, R.L., et al., 2015. Clingen—the clinical genome resource. N. Engl. J. Med. 372, 2235–2242.

Rehm, G., Moreno-Schneider, J., Bourgonje, P., 2019. Automatic and manual web annotations in an infrastructure to handle fake news and other online media phenomena. In: LREC 2018-11th International Conference on Language Resources and Evaluation. pp. 2416–2422, URL: https://www.aclweb.org/anthology/L18-1384.

Sanderson, R., 2016. Web annotation protocol. URL: https://www.w3.org/TR/annotation-protocol/.

Schön, E., Thomaschewski, J., Escalona, M.J., 2017. Agile requirements engineering: A systematic literature review. Comput. Stand. Interfaces 49, 79–91. http://dx.doi.org/10.1016/j.csi.2016.08.011.

Sein, M.K., Henfridsson, O., Purao, S., Rossi, M., Lindgren, R., 2011. Action design research. MIS Q. 35, 37–56, URL: http://misq.org/action-design-research.html.

Sjøberg, D.I.K., Anda, B., Arisholm, E., Dybå, T., Jørgensen, M., Karahasanovic, A., Koren, E.F., Vokác, M., 2002. Conducting realistic experiments in software engineering. In: 2002 International Symposium on Empirical Software Engineering. ISESE 2002, 3-4 2002, Nara, Japan, IEEE Computer Society, pp. 17–26. http://dx.doi.org/10.1109/ISESE.2002.1166921.

Tavakol, M., Dennick, R., 2011. Making sense of Cronbach's alpha. Int. J. Med. Educ. 2 (53).

Turner, M., Kitchenham, B., Brereton, P., Charters, S., Budgen, D., 2010. Does the technology acceptance model predict actual use? a systematic literature review. Inf. Softw. Technol. 52, 463–479.

Ursachi, G., Horodnic, I.A., Zait, A., 2015. How reliable are measurement scales? external factors with indirect influence on reliability estimators. Procedia Econ. Finance 20, 679–686. http://dx.doi.org/10.1016/S2212-5671(15)00123-9, globalization and Higher Education in Economics and Business Administration - GEBA 2013.

van Oordt, S., Guzman, E., 2021. On the role of user feedback in software evolution: a practitioners' perspective. In: 29th IEEE International Requirements Engineering Conference. RE 2021, Notre Dame, IN, USA, September (2021) 20-24, IEEE, pp. 221–232. http://dx.doi.org/10.1109/RE51729.2021.00027.

W3Techs, 2022. Usage statistics of traffic analysis tools for websites. URL: https://w3techs.com/technologies/overview/traffic_analysis.

Wallace, L.G., Sheetz, S.D., 2014. The adoption of software measures: A technology acceptance model (tam) perspective. Inf. Manag. 51, 249–259. http://dx.doi.org/10.1016/j.im.2013.12.003.

Wang, C., Daneva, M., van Sinderen, M., Liang, P., 2019. A systematic mapping study on crowdsourced requirements engineering using user feedback. J. Softw. Evol. Process. (31), http://dx.doi.org/10.1002/smr.2199.

Wnuk, K., Regnell, B., Karlsson, L., 2009. What happened to our features? visualization and understanding of scope change dynamics in a large-scale industrial setting. In: RE 2009 17th IEEE International Requirements Engineering Conference. Atlanta, Georgia, USA, August 31 - September 4 2009, IEEE Computer Society, pp. 89–98. http://dx.doi.org/10.1109/RE.2009.32.

Xiao, J., Zeng, J., Yao, S., Cao, Y., Jiang, Y., Wang, W., 2021. Listening to the crowd for the change file localization of mobile apps. In: ACM ICEA '21: 2021 ACM International Conference on Intelligent Computing and Its Emerging Applications. Jinan, China, December (2022) 28-29, ACM, pp. 71–76. http://dx.doi.org/10.1145/3491396.3506530.

Oscar Díaz is Full Professor at the University of the Basque Country. His current interests include Web Engineering, Web2.0, Software Product Lines, and Design Science Research. He leads a fifteen-member group, ONEKIN, with close partnership with industry. The group is being continually founded by competitive Spanish Research Programs since 1992, and it has been involved in 9 industry contracts. He obtained the B.Sc. in Computing at the University of the Basque Country in 1985, and a Ph.D. by the University of Aberdeen in 1991. He has over 100 international publications that include VLDB Journal, ACM TOIT; ACM Computing Surveys, Empirical Software Engineering or IEEE Software, and conferences such as VLDB, ICSE and WWW.

Raul Medeiros is a Ph.D. student affiliated with the Onekin Research Group at the University of the Basque Country (UPV/EHU), Spain. He holds a B.Sc. in Computer Science from the same university and an M.Sc. in Software Engineering from the University of Murcia. His current interests include Software Product Lines, Onboarding issues and Continuous Software Engineering.

Mustafa Al-Hajjaji is a senior consultant at pure-systems GmbH. He received his Ph.D. in Software Engineering in 2018 from the University of Magdeburg, Germany. His research interests are software product-line testing, model-based engineering, software testing, and mutation testing.