Semi-Automated Feature Traceability with Embedded Annotations

Hadil Abukwaik, Andreas Burger
ABB Corporate Research Center
Ladenburg, Germany

Berima Kweku Andam, Thorsten Berger
Chalmers | University of Gothenburg
Gothenburg, Sweden

***Abstract*—Engineering software amounts to implementing and evolving features. While some engineering approaches advocate the explicit use of features, developers usually do not record feature locations in software artifacts. However, when evolving or maintaining features—especially in long-living or variant-rich software with many developers—the knowledge about features and their locations quickly fades and needs to be recovered. While automated or semi-automated feature-location techniques have been proposed, their accuracy is usually too low to be useful in practice. We propose a semi-automated, machine-learning-assisted feature-traceability technique that allows developers to continuously record feature-traceability information while being supported by recommendations about missed locations. We show the accuracy of our proposed technique in a preliminary evaluation, simulating the engineering of an open-source web-application that evolved in different, cloned variants.**

***Index Terms*—software evolution, clone&own, variability, feature annotations, feature traceability, recommendation system, feature location, machine learning**

I. Introduction

Engineering software is a process of developing and evolving the features of a system. Features abstractly represent the functional and non-functional aspects of a system. They are intuitive entities understood by many different roles involved in the development. Developers, project leads, marking experts, and customers, among others, use them for communication and planning. Many different definitions of feature exist [1]. For instance, a feature can be seen as a "logical unit of behavior specified by a set of functional and non-functional requirements" [2] or as a "distinguishable characteristic of a concept (system, component, etc.) that is relevant to some stakeholder of the concept" [3]. While various software-engineering methods advocate the use of features, such as feature-driven development, features are especially important in variant-rich systems, such as software product lines [4].

Evolving and maintaining features—such as adding, removing, extending or propagating features—requires knowing the features and their (potentially scattered [5]) locations in the software artifacts. Unfortunately, recording and maintaining feature locations is tedious and error-prone [6], often requiring heavyweight tooling, such as traceability databases [7]. Traceability is even more problematic in long-living and large systems, especially when many variants exist that have been cloned [8]–[10] and need to be maintained in parallel.

Without recording and maintaining feature locations, they need to be recovered. Since manual recovery is laborious [11], [12], many automated feature-location techniques have been proposed [13]–[15]. These classify into static, dynamic or textual approaches, providing different degrees of accuracy. Unfortunately, such automated techniques are considered expensive, difficult to setup, and rather inaccurate when used in practice [13]. Therefore, it is helpful to have techniques that support traceability between features and their locations in software artifacts during system development and maintenance.

We believe that documenting features and their locations immediately and continuously during development benefits from the developers' fresh knowledge. Researchers [16]–[18] have proposed using embedded feature annotations, which evolve smoothly with the software artifacts they are embedded in. This method is cheap with low maintenance effort. As shown in a simulation study [16], continuously annotating artifacts is robust, and the benefits outweigh the costs of recording annotations, especially for variant-rich systems with many cloned variants. Yet, this method requires that developers annotate continuously and do not forget too many annotations, which typically requires an incentive for them.

We propose to semi-automate the continuous feature-annotation process with a recommender system. It learns from previous annotations done by the developers and reminds them when they forgot to annotate newly written code. As such, the recommender system is an incentive for developers to record features continuously, while it catches the cases in which the annotations are missed. To this end, we use machine learning (ML) to analyze source code changesets that are committed to the version-control system. The ML algorithm learns the system features along the commits, then it proposes the developer who commits a changeset without annotations to which features it may belong. In this case, a developer can decide to put annotations or ignore the recommendation.

We present the implementation of our recommender system and preliminary evaluation results that show a promising accuracy when simulating the history of an open source system with four cloned variants. We also present our plans of further validation on another open source system and a commercial firmware for a smart motor controller.

II. Feature Annotation Recommender System

We briefly describe the annotation system we use and how our proposed recommender system supports it.

```
1  ////&line[System Monitor]
2    void HEAPUTILModuleOp(tModOp ModOp)
3    {
4      //&begin[State Visualizer]
5      if (ModOp == CloseModOp)
6      {
7        //#12024-Remove warnings
8        // for GNU compiler
9      }
10     //&end[State Visualizer]
11   }
12 ////&line[Report Maker]
```

Listing 1. Example of embedded feature annotations (from [20])

A. Embedded Feature Annotations

We rely on the simple feature-annotation technique of Ji et al. [16]. It comprises: a syntax for feature annotations, which are put as comments into artifacts (regardless of the programming language); textual mapping files that annotate whole files or folders and which are put into the folder hierarchy; and a textual feature model in Clafer [19] syntax, which is put into the root folder. The latter just contains feature names, one per line, with indentation representing the feature hierarchy. Listing 1 shows an example of embedded annotations in source code. Using the system, the developer records feature annotations during development. For instance, when adding a feature, she adds it to the feature model and puts feature annotations around the new code. When removing a feature, she removes it from the feature model and can immediately locate its code for removal.

B. Recommender Scenario

Our recommender system supports the following traceability scenario. Developers continuously create and maintain feature traceability using the embedded feature annotations during development. These manual annotations are used by our recommender system to learn feature locations. When the developer commits new code to a version-control system, our recommender system analyzes the changeset, divides it into smaller chunks, and predicts feature locations. If a chunk is classified to belong to a feature and is not annotated as such, the recommender asks her whether she forgot to annotate the code. The developer can then decide to add annotations, thereby putting it to the exact location, or ignore the recommendation. Figure 1 illustrates this scenario, with our recommender system integrated with a version control system.

The advantage over other feature-location techniques is that the recommender learns from previous annotations, whose characteristics can be specific to the system, the development practices or the feature notion according to the developer. Note our recommender only predicts locations, not features itself. So, a new feature must be recorded by the developer with at least one location once in order to allow our recommender predicting locations in newly added code.

For machine learning algorithms, identifying features in source code is a multi-label classification problem, as a code snippet may belong to more than one feature. Thus, our feature traceability system must be able to assign multiple labels to code snippets if needed. To achieve this, each code snippet that needs to be classified or used for training the system should be formalized as we describe shortly in Sec. III-C.

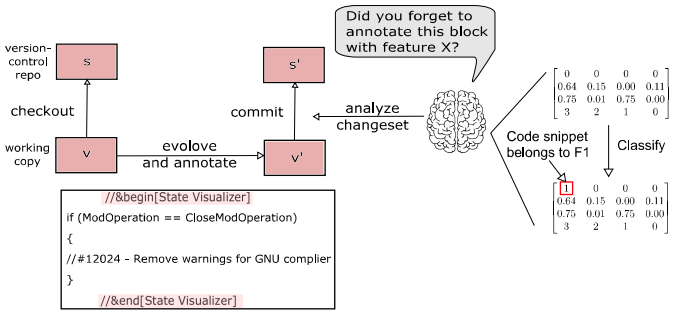

Fig. 1. Our recommender system integrated with a version-control system

III. Research Methodology

Our goal is to explore and evaluate the ML-based recommender system we proposed above. We strive to evaluate its prediction accuracy with experiments that simulate real development from the viewpoint of developers who use the recommender.

A. Research Questions

RQ1. *What is the most accurate ML classification algorithm in predicting feature locations within source code based on previous feature annotations?* We explore different ML algorithms in terms of their precision and recall in predicting features. The result is an ML classification model that is used as a base for software tools for recommending annotations.

RQ2. *How many feature-location examples must be available to give the most accurate ML prediction results?* We explore the required amount of training data for the ML classification algorithm in order to achieve an acceptable prediction accuracy. This helps to understand after how much development effort the recommender can be become effective. We investigate this question after selecting the most accurate ML algorithm with the best code granularity.

B. Subject System

We conduct experiments that simulate the development history of a variant-rich system. Specifically, we need a project that is available with its source and that has recorded features and feature annotations over its development history, which we can use as a ground truth. We select Clafer Tools [20], whose source code has been annotated with feature annotations over its whole history [16]. Clafer Tools is a portfolio of web applications written in Javascript supporting the use of the Clafer modeling language [19]. It has four variants developed using cloning. It has around 10K lines of code, 20 annotated features (i.e., the ML classification classes) and in total 14,000 feature annotations (i.e., ground truth size). We had access to its version control history consisting of commits of feature-annotated code over the development lifecycle. We could therefore replay the control version history to simulate the prediction of feature locations during development.

In the future, we also plan to simulate our proposed recommender on an industrial firmware for a smart motor controller

and further open-source systems with feature annotations [21], [22]. This will also allow us to compare our recommender system across different domains.

C. Recommender Simulation

Our experiments simulate the development history of Clafer tools until its release 0.3.5, along which developers add new code to a system upon which a trained ML algorithm predicts feature locations in the newly added code. The consecutive experiments simulate the ML performance over time as the system evolves (i.e., from the beginning of development when the system has few lines of code, to the end, when it is fully developed with thousands of lines of code).

Dataset preparation. As depicted in the upper part of Fig. 2, our data preparation includes the following steps.

Identifying feature-related code in changeset. We defined stage N of a code as its version at the Nth commit in the change repository. Thus, we identified the delta code between two consequent stages (e.g., between stage N+1 and the previous stage N). Subsequently, we extracted the feature-related code pieces from the delta using pattern matching that analyzes the embedded comments in the code that include feature names. For example, if an annotation is found in the code change set (e.g., //&line[System Monitor]), the code block containing it gets associated with the respective feature (i.e., System Monitor).

Classification granularity. This represents the level of abstraction used for classifying the feature-related code. After evaluating different abstraction levels (i.e., consecutive lines, entire files, or entire folders), we decided to chunck the changeset on the line level due to observed prediction improvement. Hence, the smallest possible chunk is a single line of code and the largest is an entire file as a block of consecutive lines. If the changeset contains an entire folder, all source code files are abstracted as a single block of lines. Afterwards, we label the chunks with the name of their related features and save them in the Feature Corpus (i.e., ground truth dataset).

Similarity calculation and classification model. Identifying the feature location in source code is considered as a multi-label classification problem, in which the set of features of the system are the classes that the ML algorithm is trained on. Accordingly, each saved block in the corpus gets formalized as a data vector D using the following metrics:

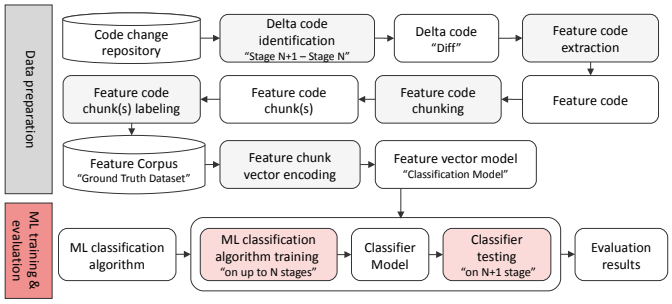

Fig. 2. ML simulation experiments

$$D = \begin{Bmatrix} \vec{f} \\ \vec{c} \\ \vec{s} \\ \vec{n} \end{Bmatrix} = \begin{bmatrix} 0 & ... & 0 \\ score_c^{F1} & ... & score_c^{Fx} \\ score_s^{F1} & ... & score_s^{Fx} \\ score_n^{F1} & ... & score_n^{Fx} \end{bmatrix}, \quad (1)$$

whereby $x = |\text{Feature}|$ is the feature name; $\vec{f}$ = Feature Presence Condition metric, which indicates if a certain code block belongs to the feature or not by 1 or 0; $\vec{c}$ = Cosine Text Similarity metric, which represents the similarity between two vectors of an inner product space that measures the cosine of the angle between them; $\vec{s}$ = Source Code Location Distance metric, which calculates the relative distance of the code block to known feature locations in the project; and $\vec{n}$ = Number of Already Existing Annotations metric, which sums up the number of existing annotations for a feature. Calculating D for each block in the corpus builds up the feature vector model that is used for training the ML algorithms.

ML training and evaluation. The ML algorithm is trained on the feature vector model and evaluated on newly added code as depicted in the lower part of Fig. 2. We repeat this process for each stage simulating the software evolution.

Experimental setup. Our experimental goal is to determine the best way to utilize the ML capability in predicting features in source code. Hence, we run a set of experiments, in which we alternate different well-known binary classification ML algorithms including Support Vector Machine (SVM), k-Nearest Neighbor (kNN), and Decision Tree (DT).

Evaluation method and metrics. Each ML classification algorithm is trained on the feature vector model of stage N results in a classifier model that we test on data vectors for new code blocks from stage N+1. As seen in the right of Fig. 1, the input for a classifier test is a new set of data vectors for an unclassified code blocks. The classifier output then indicates to which features a code block may belong to. For each stage in the code change repository, we run the designed experiments and record the performance for each algorithm on the different classification granularities. The overall performance of each algorithm is calculated as an average of its individual scores.

To quantify the performance, we use the well-known classification accuracy metrics: precision (i.e., the ratio of correctly identified and labeled code blocks by the algorithm to the total number of code blocks it identifies and labels), recall (i.e., the ratio of correctly identified and labeled code blocks by the algorithm to the total number of existing code blocks) and the F-measure (i.e., the harmonic mean of precision and recall that is calculated as: (2*Precision*Recall)/(Precision+Recall)).

IV. Emergent Results and Discussion

We now report the emergent results of the first executed case and postulate answers to the research questions.

Execution implementation. We developed a Java tool that performs the data pre-processing, generates the ML classification model, and creates the evaluation results. The tool reuses information retrieval libraries from the Lucene search engine and ML libraries from the WEKA toolkit.

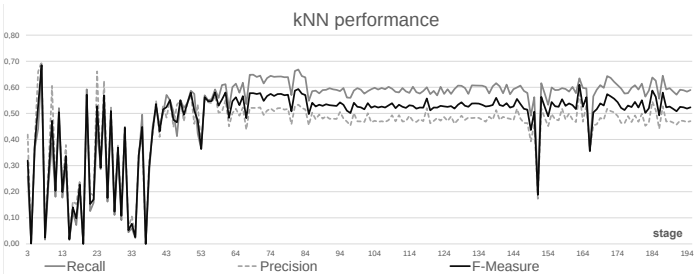

Fig. 3. Accuracy of kNN for the simulated development stages

RQ1 and RQ2. The best performing algorithm is the kNN, in which the F-measure ranges between 50 % and 60 % after a certain training phase (see Fig. 3). In fact, the accepted classification accuracy depends on the domain. For reminding developers about missing annotations in changesets of source code, we believe 60 % accuracy is good, especially that there are influencing factors that are out of control and further information unavailable (e.g., the experience of the developer who writes the code, the complexity of the implemented feature, the dependency between features, etc.). The drops in the kNN classification accuracy, after reaching a stable range, are due to major code refactorings that introduced new features that required more training for the algorithm. As seen, the size of the initial training set required for achieving stable range of prediction results for the kNN algorithm is about 60 commits with an average of 5 annotations or data vectors each. For the SVM algorithm, the prediction results are not promising, as the F-measure stables at a range between around 7 % and 15 %. The DT algorithm results are even lower, making it unusable.

Discussion. Despite the promising results of the preliminary case study, we plan to validate the method more thoroughly. The data preparation stage, in this case, included code blocks of annotated features only (40 % of the total code). However, in real life, there are code blocks that do not belong to any feature (e.g., system code or configuration code). Accordingly, these blocks will be raised up frequently by the recommender that will try to classify them under irrelevant features. Hence, we plan to extend the ground truth dataset with code blocks that do not belong to features and label them with a new label (e.g., "common code"). Then, we plan to retrain the ML algorithm on the extended dataset and retest the feature prediction accuracy. We expect this to significantly improve our results.

Our recommender expects that every ML prediction is assessed by the person who commits the code change. So far, the required assessment frequency is not clear. To determine adequate frequency, it may be necessary to evaluate the recommender in an industrial case. Indeed, for the first number of commits (∼70% in our case study), the recommender requires more assessments, as the system is still in the learning phase and with a lot of false positives. Nevertheless, the assessment frequency is expected to decrease over time as reminding developers frequently to add feature tags will decrease the number of cases in which they forget to annotate.

Moreover, our approach is sensitive to the content of the ground truth dataset (i.e., the existing examples of labeled source code blocks). Therefore, we plan to literally replicate the case study on three unique cases, outlined in II. This facilitates obtaining more significant results and drawing generalizable conclusions on the cross-case level. Although, the first case results show that our approach works with room for improvements, replicating cases brings up more observations about the approach potentials and limitations.

In our experiments, we used the default parameter values for all algorithms. Accordingly, we see a potential improvement for the prediction results by tuning the parameters of the algorithms. We also did not optimize the code chunks extracted from the change dataset repository. Thus, we see more optimization potential by removing programming-language-specific syntax.

V. Related Work

Many feature location techniques were conceived: static [6], [23], [24], dynamic [25] and textual techniques [26], [27]. Additionally, there are techniques that use textual similarity, analyze static dependencies or use call graphs [6], [15], [23], [24], [26], and program traces [25]–[28]. A good overview on these different techniques is provided by Cornelissen et al. [25] and Robillard [23]. These techniques try to retroactively recover features or feature locations—information in the developers' minds during development that was lost. As a survey shows [13] it is intrinsically difficult to automatically recover this information. We propose a different route: We encourage developers to continuously record features and we support them with a traceability system that learns from previous annotations.

Ji et al. [16] propose feature traceability using embedded annotations. We build our recommender upon theri work by using their annotation system for recording feature locations.

LEADT [29] is an Eclipse plug-in used to manually annotate variable code. LEADT also provides a kind of recommender system for feature location in legacy code, yet, its use case is different. It focuses on retroactive feature location in legacy code by exploiting the extraction of feature dependencies. LEADT supports only JAVA, while our feature recommendation system is programming-language-independent. LEADT focuses solely on optional features (i.e., when a product line is adopted) while we strive to support single-system development and clone&own-based development for many variants. Of course, the feature annotations are especially helpful for adopting a product line by integrating the cloned variants [16].

The Suade tool [30] offers suggestions for software investigations. Developers trigger recommendations on what code elements to investigate among all related ones. They explicitly specify the set of relevant fields and methods (i.e., the context elements), and Suade uses method-call and field-access relations to automatically retrieve related elements. It finally ranks the retrieved elements by extracting a dependency graph of all their static dependencies from the source code to the context elements. In our approach, developers do not need manual triggering for the system as it acts autonomously during development. Also, developers do not specify the context as the system learns the different features independently.

Guo et al. present an approach [31] that focuses on deep learning (an Recurrent Neural Network model) to incorporate requirements artifact semantics and domain knowledge into the tracing solution. This approach does not take into account an interactive feedback-driven step. It is also shown that neural networks are not fast enough to provide instant feedback to developers once they forget to annotate a code snippet.

Recommendation systems are widely used. Robillard et al. [32] and Pakdeetrakulwong et al. [33] summarize and categorize some of the available approaches, finding that all listed recommendation systems do not focus on feature identification, tracing, or documentation. Instead, they recommend associated requirements, similar source code snippets, bug-fixing suggestions, or comments published on internet forums related to a particular source code by other developers. None of these recommendation systems addresses our use case.

VI. Conclusion

We proposed a feature-traceability recommender system that can be integrated seamlessly in an off-the-shelf version control system. It relies on the idea of tagging source code with feature annotations continuously during development [16], enabling traceability between the implementation artifacts and the features of the system. By analyzing and formalizing the different annotated source code blocks, the traceability system autonomously learns the features of the software system.

The feature traceability recommender is not a fully automatic solution for tagging source code with feature information, but it provides recommendations on the feature affinity for new source code blocks once a developer forgets to annotate it. Our preliminary case study shows promising results, even though, our implementation does not contain all possible optimizations. We reached an overall accuracy of about 50 %. In the future, we plan for more case studies with optimizations for the configurations of the used machine learning algorithm and the handling of the source code blocks. Additionally, evaluating the technique acceptance by practitioners in terms of usefulness and ease-of-use will be taken into account.

Acknowledgment

This work is supported by the ITEA project REVaMP^2 and funded by the German Federal Ministry of Education and Research (01IS16042B), by Vinnova Sweden (2016-02804), and the Swedish Research Council Vetenskapsrådet (257822902).

References

[1] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki, "What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines," in *SPLC*, 2015.

[2] J. Bosch, *Design and use of software architectures: adopting and evolving a product-line approach*. Pearson Education, 2000.

[3] K. Czarnecki and U. W. Eisenecker, *Generative programming: methods, tools, and applications*. Addison Wesley, 2000.

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

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

[6] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, "The concept assignment problem in program understanding," in *ICSE*, 1993.

[7] S. Winkler and J. von Pilgrim, "A survey of traceability in requirements engineering and model-driven development," *Software & Systems Modeling*, vol. 9, no. 4, pp. 529–565, 2010.

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

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

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

[11] J. Krüger, T. Berger, and T. Leich, *Features and How to Find Them: A Survey of Manual Feature Location*. LLC/CRC Press, 2018.

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

[13] J. Rubin and M. Chechik, "A survey of feature location techniques," in *Domain Engineering*, 2013.

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

[15] A. Burger and S. Gruner, "Finalist 2: Feature identification, localization, and tracing tool," in *SANER*, 2018.

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

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

[18] M. Seiler and B. Paech, "Using tags to support feature management across issue tracking systems and version control systems," in *REFSQ*, 2017.

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

[20] M. Antkiewicz, K. Bąk, A. Murashkin, R. Olaechea, J. Liang, and K. Czarnecki, "Clafer tools for product line engineering," in *SPLC*, 2013.

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

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

[23] M. P. Robillard, "Topology analysis of software dependencies," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 17, no. 4, p. 18, 2008.

[24] M. Petrenko and V. Rajlich, "Variable granularity for improving precision of impact analysis," in *ICPC*, 2009.

[25] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 684–702, 2009.

[26] M. Eaddy, A. V. Aho, G. Antoniol, and Y.-G. Guéhéneuc, "Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis," in *ICPC*, 2008.

[27] T. Savage, M. Revelle, and D. Poshyvanyk, "Flat 3: feature location and textual tracing tool," in *ICSE*, 2010.

[28] D. Simon and T. Eisenbarth, "Evolutionary introduction of software product lines," in *SPLC*, 2002.

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

[30] F. W. Warr and M. P. Robillard, "Suade: Topology-based searches for software investigation," in *ICSE*, 2007.

[31] J. Guo, J. Cheng, and J. Cleland-Huang, "Semantically enhanced software traceability using deep learning techniques," in *ICSE*, 2017.

[32] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *IEEE Software*, vol. 27, pp. 80–86, 2010.

[33] U. Pakdeetrakulwong, P. Wongthongtham, and W. V. Siricharoen, "Recommendation systems for software engineering: A survey from software development life cycle phase perspective," in *ICITST*, 2014.