Comparison-Based Feature Location in ArgoUML Variants

Gabriela Karoline Michelon[1,2], Lukas Linsbauer[1,3], Wesley K. G. Assunção[4], Alexander Egyed[1]
[1]Institute for Software Systems Engineering - Johannes Kepler University Linz - Austria
[2]LIT Secure and Correct Systems Lab - Johannes Kepler University Linz - Austria
[3]Christian Doppler Laboratory MEVSS - Johannes Kepler University Linz - Austria
[4]COTSI - Federal University of Technology - Paraná, Toledo, Brazil
gabriela.michelon@jku.at,lukas.linsbauer@jku.at,wesleyk@utfpr.edu.br,alexander.egyed@jku.at

ABSTRACT

Identifying and extracting parts of a system's implementation for reuse is an important task for re-engineering system variants into Software Product Lines (SPLs). An SPL is an approach that enables systematic reuse of existing assets across related product variants. The re-engineering process to adopt an SPL from a set of individual variants starts with the location of features and their implementation, to be extracted and migrated into an SPL and reused in new variants. Therefore, feature location is of fundamental importance to the success in the adoption of SPLs. Despite its importance, existing feature location techniques struggle with huge, complex, and numerous system artifacts. This is the scenario of ArgoUML-SPL, which stands out as the most used case study for the validation of feature location approaches. In this paper we use an automated feature location technique and apply it to the ArgoUML feature location challenge posed.

CCS CONCEPTS

• **Software and its engineering** → **Software product lines**; **Traceability**; **Software reverse engineering**; *Reusability*.

KEYWORDS

feature location, traceability, variants, clones, reuse, software product lines

ACM Reference Format:
Gabriela Karoline Michelon[1,2], Lukas Linsbauer[1,3], Wesley K. G. Assunção[4], Alexander Egyed[1]. 2019. Comparison-Based Feature Location in ArgoUML Variants. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3336294.3342360

1 INTRODUCTION

To conduct software maintenance or evolution, feature location is one of the key steps [17]. The goal of feature location is to establish mappings/traces between features and their respective implementation. Feature location allows practitioners to find and reason about the parts of a system that will be affected by modifications [6]. Many feature location techniques can be found in literature [17]. In addition, an increasing interest and proliferation of techniques are observed in the field of re-engineering variants, developed using ad hoc reuse, into Software Product Lines (SPLs) [2, 3]. To evaluate these techniques, ArgoUML-SPL is one of the most used subject system, as shown in the ESPLA catalog [13].

Martinez et al. propose a challenge case study based on ArgoUML-SPL at the Systems and Software Product Line Conference (SPLC) [14]. This challenge defines the ArgoUML-SPL benchmark that is comprised of eight optional features and 15 predefined scenarios ranging from a single variant to 256 variants. The benchmark provides a set of tools and artifacts, including a ground truth to allow techniques to be evaluated and compared to each other. The main challenging characteristics of ArgoUML-SPL are: (i) it is a real software system of considerable size that resulted from a development process involving several developers; (ii) the implementation is composed of feature interactions and feature negations; (iii) the granularity of the implementation that must be traced varies from complete Java classes to statements inside methods.

Taking into account the challenges aforementioned, this work applies our feature location technique implemented in the ECCO tool [7, 11, 12] to the ArgoUML-SPL challenge [14] and presents and discusses the results. The contributions of this study are: (i) implementation of a Java adapter that extracts data from the Java source code files at the granularity asked for by the challenge (classes, methods and their respective refinements); (ii) application of ECCO to a real-world subject where we can show how well our approach deals with these system variants; (iii) providing and discussing results for the ArgoUML challenge case study so that others can compare their results to ours.

The remainder of this paper is structured as follows: Section 2 discusses related work. Section 3 briefly describes the challenge dataset. Section 4 describes our feature location technique and the methodology used to apply it to the posed challenge. Section 5 presents the results and offers a discussion with particularly interesting insights. Finally, Section 6 concludes.

2 RELATED WORK

Feature location is the first step in the process of re-engineering variants into SPLs [2]. To ease the migration of software variants into an SPL, some feature location techniques can automate the location of source code elements relevant to a given feature. ArgoUML-SPL is the subject system most used in this context [13].

Rubin and Chechik [17] describe the most common feature location techniques and categorize them by the strategies they use:

SPLC '19, September 9–13, 2019, Paris, France

ACM ISBN 978-1-4503-7138-4/19/09…$15.00
https://doi.org/10.1145/3336294.3342360

static program analysis, which leverages static dependencies between program elements; information retrieval techniques, based on information embedded in program identifier names and comments; and dynamic approaches that collect precise information about the program execution. In addition, hybrid approaches combine different techniques to take advantage of each approach.

Cruz et al. [5] provide a literature review and use ArgoUML-SPL to evaluate three information retrieval based feature location techniques. These strategies were compared based on their ability to correctly identify the source code of several features from the ArgoUML-SPL ground truth. The results suggest that Latent Semantic Indexing is better than both Paragraph Vectors and Latent Dirichlet Allocation. However, the values obtained for precision, recall and F1 measure were not satisfactory, and strategies to better deal with huge and complex artifacts are needed.

There are several comparison-based feature identification (e.g. [22]) or feature location (e.g. [1, 18, 21, 23]) techniques. Most of them rely on formal concept analysis (e.g. [1, 18, 21]). Many of them are hybrid approaches based on a combination of formal concept analysis and information retrieval techniques (e.g. [18, 21]) of which our approach uses neither. Most only consider single features and not their negations or interactions (i.e. conjunctions or disjunctions), can only be applied to specific types of implementation artifacts (e.g. source code), or only operate on a coarse level of implementation artifacts (e.g. class or method level and not statement level). Our approach has none of these restrictions.

Martinez et al. built the tool BUT4Reuse [15, 16]. BUT4Reuse supports extractive SPL adoption by providing a framework and techniques for feature identification, feature location, mining feature constraints, extraction of reusable assets, feature model synthesis and visualizations, etc. This tool is designed to be generic and extensible, allowing researchers to include their own strategies. To this end, they also employ an adapter concept in order to support variability in different types of implementation artifacts.

3 DATA SET

The ArgoUML-SPL challenge [14] provides *15 scenarios* and a *ground truth* consisting of *24 traces*. ArgoUML is an open source tool for UML modeling that is implemented in Java and was refactored into a product line [4]. It consists of *two mandatory features*: Diagrams Core, and Class Diagram; and *eight optional features*: State Diagram, Activity Diagram, Use Case Diagram, Collaboration Diagram, Deployment Diagram, Sequence Diagram, Cognitive Support, and Logging.

3.1 Scenarios

The challenge provides *15 scenarios* (see Table 1) with varying number of variants. In our context, a variant is defined as follows:

Definition. *A* variant *V is a pair (F, A) that maps a set of features F that the variant provides to a set of implementation artifacts A that implement the variant.*

More specifically, *Original* scenario contains only the initial ArgoUML system as a single variant from which the ArgoUML-SPL was initially created [4]. *Traditional* scenario has 10 variants, one with all the features, one with only the mandatory features, and for every optional feature one variant with only that feature disabled.

Table 1: ArgoUML Challenge Scenarios

Scenario	Size	Description
Original	1	Original ArgoUML variant containing all features.
Traditional	10	Variants with no, all, and combinations of 7 optional features.
PairWise	9	Set of variants that covers all pairwise feature combinations.
2-10 Random	2-10	Randomly selected subsets of variants.
50 Random	50	Randomly selected subset of variants.
100 Random	100	Randomly selected subset of variants.
All	256	All possible variants of ArgoUML-SPL.

PairWise scenario is composed of 9 variants obtained using the pairwise feature coverage algorithm from FeatureIDE [20]. *Random* scenarios of different size (2, 3, 4, 5, 6, 7, 8, 9, 10, 50 and 100 variants) with randomly selected variants. *All* scenario has all the $2^8 = 256$ possible variants of ArgoUML-SPL obtained from FeatureIDE [20].

3.2 Ground Truth

The ground truth provided by the challenge consists of *24 traces*.

Definition. *A* trace *T is a pair (F, A) that maps a propositional logic formula F whose literals are features to a set of implementation artifacts A.*

In the context of this particular challenge, implementation artifacts represent Java code elements. In other words, every trace T maps a set of code elements $T.A$ to a feature condition $T.F$.

The ground truth contains one trace for each of the eight individual features; Two traces with a single negative feature; 13 traces with a conjunction of two features; One trace with a conjunction of three features.

4 FEATURE LOCATION TECHNIQUE

We applied an automatic feature location technique that is based on the comparison of features and implementation artifacts of a set of variants [7, 10–12]. The approach was first presented at SPLC'13 [10] and has since evolved beyond the task of feature location (i.e. trace computation) to also support, for example, the composition of variants from the computed traces [9, 12]. It is now implemented in a publicly available tool called ECCO[1,2] [7, 12]. ECCO supports trace extraction (*commit* operation) and variant composition (*checkout* operation) with arbitrary types of implementation artifacts beyond just source code. To this end, it requires an adapter for every type of artifact that contains variability. To apply it to the posed challenge we developed a simple Java adapter to parse the code using JavaParser[3,4] [19].

Figure 1 shows the feature location process of ECCO. The entire implementation of a variant (in this case a set of Java files) is input to the responsible adapter (in this case the Java adapter). The adapter creates an artifact tree structure as illustrated in Figure 2. The *commit* operation uses the artifact tree structure and the set of features of the variant to compute traces from features to nodes in the artifact tree, which are finally stored in a repository. This process can be repeated incrementally with arbitrarily many variants. Every time a new variant is committed the traces in the repository

[1] https://jku-isse.github.io/ecco/
[2] https://github.com/jku-isse/ecco/
[3] https://javaparser.org/
[4] https://github.com/javaparser

are refined. One could, for example, decide to stop committing further variants when the traces in the repository have not changed anymore over the last few commits. Finally, the traces stored in the repository are exported into the format requested by the challenge and compared to the ground truth traces (see Section 3.2) to compute the challenge metrics (see Section 5.2).

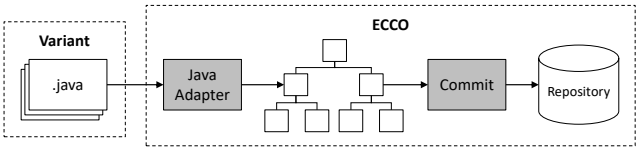

Figure 1: ECCO process for committing a single variant consisting of multiple Java files into its repository.

Figure 2 illustrates the structure of the artifact tree created from Java files by the Java adapter. It reflects the requirements of the challenge that asks to trace classes, methods and their respective refinements in the form of changes to imports, fields and lines of code. Therefore, we store classes (including nested classes), imports, fields and methods explicitly. For fields and methods we also store their children (to be able to detect refinements) in the form of raw lines of code as they can span over multiple lines. This is necessary as variability in ArgoUML-SPL is often implemented at such a fine-granular level and annotations are not always placed in a disciplined manner [8]. Even single statements, e.g. field declarations, can span multiple lines of which only some might be annotated with features.

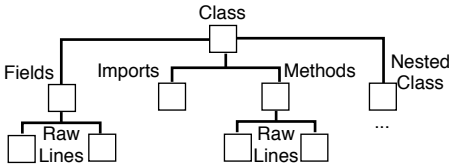

Figure 2: Artifact tree structure created by the Java adapter.

The *commit* operation is essentially based on five rules [11]. Assume two variants A and B:

(1) Common artifacts (in A and B) *likely* trace to common features (in A and B).
(2) Artifacts in A and not B *likely* trace to features in A and not B, and vice versa.
(3) Artifacts in A and not B *do not* trace to features in B and not A, and vice versa.
(4) Artifacts in A and not B *at most* trace to features in A, and vice versa.
(5) Artifacts in A and B *at most* trace to features in A or B.

The first two rules quickly isolate features (or feature combinations) to which implementation artifacts likely trace. Rule 3 determines features to which implementation artifacts certainly cannot trace. The last two rules provide an upper bound on where features can at most trace. For more details please consult [11].

5 RESULTS

This section presents and discusses the results of our feature location technique ECCO being applied to each of the 15 scenarios. The results and instructions for reproducing them are publicly available[5].

5.1 Time Performance

Figure 3 shows a box plot of the runtime per variant (i.e. *commit* operation) per scenario, ordered by increasing number of variants. It was measured on an HP ZBook 14 laptop, with Intel® Core™ i7-4600U processor (2.1GHz, 2 cores), 16GB of RAM and SSD storage, running Fedora Linux as operating system. Each scenario's average runtime per committed variant is between 4 and 7 seconds. Overall, the runtime remains quite constant. Fluctuations are most likely caused by differences in the size of variants.

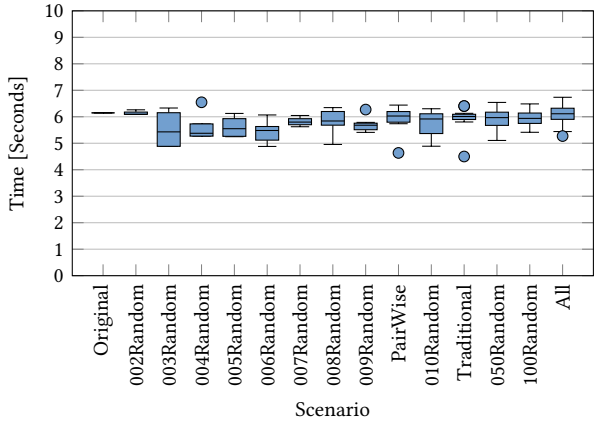

Figure 3: Runtime per Variant per Scenario

5.2 Precision, Recall, F1 Score

To validate the computed traces, three metrics are computed *for each trace T*. They are calculated automatically by the challenge benchmark [14], which compares every ground truth trace T_{gt} with the respective (i.e. same feature condition $T.F$) trace T_{ecco} computed by our feature location technique.

Precision is the percentage of correctly retrieved artifacts (i.e. code elements) relative to the total retrieved code elements.

$$precision = \frac{TP}{TP + FP} = \frac{|T_{gt}.A \cap T_{ecco}.A|}{|T_{ecco}.A|} \quad (1)$$

where TP (true positives) are the correctly retrieved code elements and FP (false positives) are the incorrectly retrieved code elements.

Recall is the percentage of correctly retrieved code elements relative to the total number of code elements in the ground truth.

$$recall = \frac{TP}{TP + FN} = \frac{|T_{gt}.A \cap T_{ecco}.A|}{|T_{gt}.A|} \quad (2)$$

where FN (false negatives) are code elements present in the ground truth which are not included in the retrieved code elements.

[5] https://github.com/jku-isse/SPLC2019-Challenge-ArgoUML-FeatureLocation

The F1 score (F-measure) relates precision and recall and combines them into a single measure.

$$F1 = 2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}} \quad (3)$$

Figure 4 shows average precision, recall and F1 score over all traces per scenario, ordered by increasing number of variants.

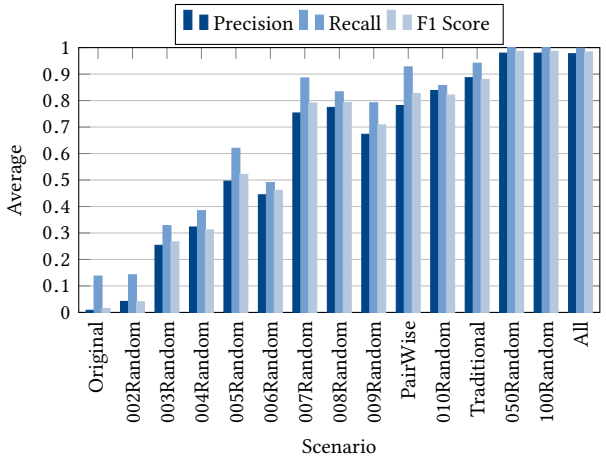

Figure 4: Average Precision, Recall and F1 Score per Scenario

The best results were obtained with the largest scenarios which achieved 100% average recall and around 99% average precision and F1 score. Overall, the results of our feature location technique improve the more variants are available. This is to be expected, as our technique is based on the comparison (i.e. commonalities and differences) of features and implementation of variants. As a consequence of this, scenarios consisting of only one or two variants produce quite useless results.

However, it is not generally true that more variants always produce better results. For example, the 9 random variants produce a slightly worse result than the 8 random variants. Similarly, the 10 traditional variants produce a slightly better result than the 10 random variants. This means that variants with *beneficial* configurations (i.e. combinations of features that exhibit *interesting* variability) can make up for a lower number of total variants available. Also, there seems to be a critical point after which the results do not improve much anymore, which is somewhere around 10 variants. After that, every additional variant only improves the results marginally. Beyond 50 variants the results do not even change at all anymore as nothing new can be learned from additional variants.

The few differences that remain in the results, even with all variants available for analysis, are caused by ambiguities during the alignment of sequences of lines of source code (for example, children of methods in the artifact tree, see Figure 2) during the comparison of the implementation of variants. Alignments of sequences of lines, i.e. insertions and deletions, do not always reflect perfectly the actual changes that were performed, as anyone who has ever used a source code diffing tool, e.g. when performing merges in a VCS such as Git, is probably aware. This causes misinterpretations of changes and leads to mismatches with the ground truth in some cases. However, this does not mean that the computed traces are *wrong* as they still produce the exact same variants, it just means that there are multiple valid traces and that the one our approach computed does not match the one provided by the ground truth. This is illustrated with a minimalistic example in Figure 5. It shows three variants with features $\{A\}$, $\{A, B\}$ and $\{B\}$ respectively and alignments of the statement `i++` on the left and the corresponding traces that produce the variants in the form of annotated code on the right. The two rows illustrate two different alignments and corresponding different traces. Even though the traces (on the right) are different, the produced variants (on the left) are identical.

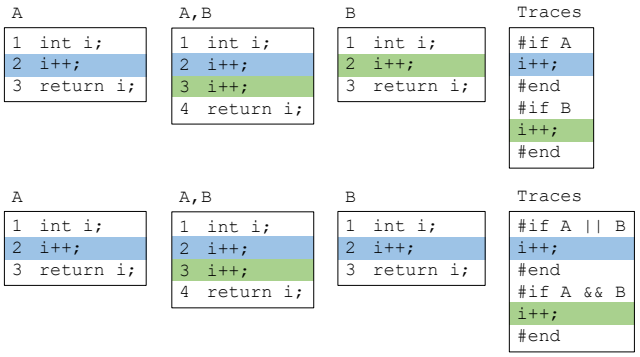

Figure 5: Illustration of two valid alignments of lines (top and bottom) in three variants with two optional features A and B (left side) and the corresponding traces (right side).

6 CONCLUSION

This work presents a solution to the ArgoUML-SPL feature location challenge posed at SPLC [14]. We applied ECCO, an automatic feature location technique, that is based on the comparison of features and implementation of a set of variants. Our technique computes a set of traces that map features (actually propositional logic formulas with features as literals) to code elements. Overall, the more variants are available the better our computed traces match the ground truth traces provided by the challenge. However, a critical point is reached somewhere around 10 variants where precision and recall reach around 90% and after that they only improve marginally with every additional variant. The highest gain per additional variant is achieved with the first few variants. The runtime of our technique increases linearly with the number of variants.

ACKNOWLEDGMENTS

This research was funded by the JKU Linz Institute of Technology (LIT) and the state of Upper Austria, grant no. LIT-2016-2-SEE-019; the LIT Secure and Correct Systems Lab funded by the state of Upper Austria; the Austrian Science Fund (FWF), grant no. P31989; the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and KEBA AG, Austria; Pro2Future, a COMET K1-Centre of the Austrian Research Promotion Agency (FFG), grant no. 854184; and CNPq grant 408356/2018-9.

REFERENCES

[1] Ra'Fat Al-Msie'deen, Abdelhak Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Hamzeh Eyal Salman. 2013. Feature Location in a Collection of Software Product Variants Using Formal Concept Analysis. In *Safe and Secure Software Reuse (ICSR)*, Vol. 7925. Springer, Berlin, Heidelberg, 302–307. https://doi.org/10.1007/978-3-642-38977-1_22

[2] Wesley K. G. Assunção, Roberto E. Lopez-Herrejon, Lukas Linsbauer, Silvia R. Vergilio, and Alexander Egyed. 2017. Reengineering legacy applications into software product lines: a systematic mapping. *Empirical Software Engineering* 22, 6 (Dec 2017), 2972–3016. https://doi.org/10.1007/s10664-017-9499-z

[3] Wesley K. G. Assunção and Silvia R. Vergilio. 2014. Feature location for software product line migration: a mapping study. In *18th International Software Product Lines Conference - Companion Volume for Workshop, Tools and Demo papers (SPLC '14)*. ACM, Florence, Italy, 52–59. https://doi.org/10.1145/2647908.2655967

[4] Marcus Vinicius Couto, Marco Tulio Valente, and Eduardo Figueiredo. 2011. Extracting Software Product Lines: A Case Study Using Conditional Compilation. In *15th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, Oldenburg, Germany, 191–200. https://doi.org/10.1109/CSMR.2011.25

[5] Daniel Cruz, Eduardo Figueiredo, and Jabier Martinez. 2019. A Literature Review and Comparison of Three Feature Location Techniques using ArgoUML-SPL. In *13th International Workshop on Variability Modelling of Software-Intensive Systems*. ACM, 16.

[6] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process* 25 (2013), 53–95. https://doi.org/10.1002/smr.567

[7] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2015. The ECCO Tool: Extraction and Composition for Clone-and-Own. In *37th IEEE/ACM International Conference on Software Engineering*, Vol. 2. IEEE Computer Society, Florence, Italy, 665–668. https://doi.org/10.1109/ICSE.2015.218

[8] Jörg Liebig, Christian Kästner, and Sven Apel. 2011. Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In *10th International Conference on Aspect-Oriented Software Development, AOSD 2011, Porto de Galinhas, Brazil, March 21-25, 2011*, Paulo Borba and Shigeru Chiba (Eds.). ACM, 191–202. https://doi.org/10.1145/1960275.1960299

[9] Lukas Linsbauer, Alexander Egyed, and Roberto Erick Lopez-Herrejon. 2016. A variability aware configuration management and revision control platform. In *38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, Laura K. Dillon, Willem Visser, and Laurie Williams (Eds.). ACM, 803–806. https://doi.org/10.1145/2889160.2889262

[10] Lukas Linsbauer, E. Roberto Lopez-Herrejon, and Alexander Egyed. 2013. Recovering Traceability Between Features and Code in Product Variants. In *17th International Software Product Line Conference (SPLC '13)*. ACM, New York, NY, USA, 131–140. https://doi.org/10.1145/2491627.2491630

[11] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2017. Variability extraction and modeling for product variants. *Software & Systems Modeling* 16, 4 (Oct 2017), 1179–1199. https://doi.org/10.1007/s10270-015-0512-y

[12] Stefan Fischer Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *IEEE International Conference on Software Maintenance and Evolution*. IEEE Computer Society, Victoria, BC, Canada, 391–400. https://doi.org/10.1109/ICSME.2014.61

[13] Jabier Martinez, Wesley K. G. Assunção, and Tewfik Ziadi. 2017. ESPLA: A Catalog of Extractive SPL Adoption Case Studies. In *21st International Systems and Software Product Line Conference - Volume B (SPLC '17)*. ACM, New York, NY, USA, 38–41. https://doi.org/10.1145/3109729.3109748

[14] Jabier Martinez, Nicolas Ordoñez, Xhevahire Tërnava, Tewfik Ziadi, Jairo Aponte, Eduardo Figueiredo, and Marco Tulio Valente. 2018. Feature Location Benchmark with ArgoUML SPL. In *22Nd International Systems and Software Product Line Conference - Volume 1 (SPLC '18)*. ACM, New York, NY, USA, 257–263. https://doi.org/10.1145/3233027.3236402

[15] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2015. Bottom-up adoption of software product lines: a generic and extensible approach. In *19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, Douglas C. Schmidt (Ed.). ACM, 101–110. https://doi.org/10.1145/2791060.2791086

[16] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2017. Bottom-up technologies for reuse: automated extractive adoption of software product lines. In *39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE Computer Society, 67–70. https://doi.org/10.1109/ICSE-C.2017.15

[17] Julia Rubin and Marsha Chechik. 2013. A Survey of Feature Location Techniques. In *Domain Engineering, Product Lines, Languages, and Conceptual Models*. Springer, Berlin, Heidelberg, 1–51. https://doi.org/10.1007/978-3-642-36654-3_2

[18] Hamzeh Eyal Salman, Abdelhak-Djamel Seriai, and Christophe Dony. 2013. Feature-to-code traceability in a collection of software variants: Combining formal concept analysis and information retrieval. In *IEEE 14th International Conference on Information Reuse & Integration, IRI 2013, San Francisco, CA, USA, August 14-16, 2013*. IEEE Computer Society, 209–216. https://doi.org/10.1109/IRI.2013.6642474

[19] Nicholas Smith, Danny van Bruggen, and Federico Tomassetti. 2018. *JavaParser for Processing Java Code*. https://javaparser.org/

[20] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming* 79 (2014), 70–85. https://doi.org/10.1016/j.scico.2012.06.002

[21] Yinxing Xue, Zhenchang Xing, and Stan Jarzabek. 2012. Feature Location in a Collection of Product Variants. In *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*. IEEE Computer Society, 145–154. https://doi.org/10.1109/WCRE.2012.24

[22] Tewfik Ziadi, Luz Frias, Marcos Aurélio Almeida da Silva, and Mikal Ziane. 2012. Feature Identification from the Source Code of Product Variants. In *16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012*, Tom Mens, Anthony Cleve, and Rudolf Ferenc (Eds.). IEEE Computer Society, 417–422. https://doi.org/10.1109/CSMR.2012.52

[23] Tewfik Ziadi, Christopher Henard, Mike Papadakis, Mikal Ziane, and Yves Le Traon. 2014. Towards a language-independent approach for reverse-engineering of software product lines. In *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*, Yookun Cho, Sung Y. Shin, Sang-Wook Kim, Chih-Cheng Hung, and Jiman Hong (Eds.). ACM, 1064–1071. https://doi.org/10.1145/2554850.2554874