Software Product Line Extraction from Bytecode based applications

Tewfik Ziadi, Lom Messan Hillah

Software Product Line Extraction from Bytecode based applications

Tewfik Ziadi
Sorbonne Université
CNRS, LIP6
F-75005 Paris, France
Email: Tewfik.Ziadi@lip6.fr

Lom Messan Hillah
Univ. Paris Nanterre & Sorbonne Université
CNRS, LIP6
F-75005 Paris, France
Email: Lom-Messan.Hillah@lip6.fr

***Abstract*—This paper presents a Software Product Line (SPL) extraction approach to handle legacy software systems running on the Java Virtual Machine (JVM), for which the source code is unavailable, and factor in all input programming languages for the JVM. The approach extracts from the bytecode of a collection of software variants created using the Clone-And-Own approach a full SPL with its feature model. We provide a full implementation and integration in the BUT4Reuse framework. An early experiment was carried out on the ArgoUML case study and preliminary results are discussed.**

***Index Terms*—Software Product Line, Bytecode, Analysis**

I. Introduction

Software Product Lines Engineering (SPLE) [1] defines two phases: *domain* and *application* engineering [2]. Domain engineering defines the scope of the SPL, and specifies the *commonality* and *variability* among products using the concepts of *features* and *feature models* [2]. A feature is a prominent or distinctive characteristic, quality or user-visible aspect of a software system or systems [3]. Feature models (FM) are popular in SPLE to describe both variability and commonalities in a family of product variants [4]. Figure 1 shows a feature model illustrating a simplified example of an electronic shop. The `E-Shop` FM consists of a mandatory feature `Catalogue`, two possible `Payment` methods from which one or both could be selected, an alternative between two `Security` levels, and an optional `Search` feature.

Adopting an SPL approach and designing SPL variability is a major challenge for companies [5], [6]. Berger *et al*. showed that around 50% of companies participating in industrial SPL Engineering cannot adopt SPL proactively [7]. On the one hand, the variability is discovered as customers' needs emerge over time; so, it is tough, if not impossible, to anticipate all the variations from the beginning. On the other hand, companies already have created product variants using opportunistic ad-hoc reuse to quickly respond to different customers' needs. As a consequence, instead of adopting an SPL, many companies clone an existing product and modify it to fit new customer needs [8]. This approach, called *Clone-and-Own*, is popular because it is faster and more efficient to start with an already developed and tested set of artifacts [8].

The *Clone-and-Own* approach incurs a severe drawback which is the high cost of managing and maintaining many variants at the same time [7]. For instance, when a bug is identified and fixed in one variant, the question of propagating the bug fix onto all existing variants becomes very difficult and error-prone when the number of variants increases. One solution to deal with this problem is to migrate these existing variants into an SPL.

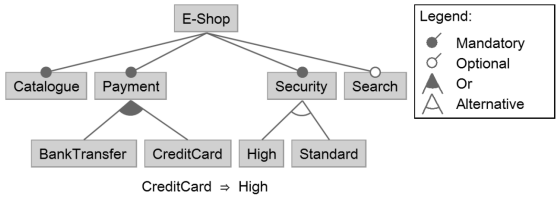

Fig. 1. E-Shop feature model

Although the migration is a well-known challenge [9], several approaches, referred to as *extractive* or *bottom-up* approaches, have been proposed for source code artifacts [10], [11] or models [12]. In this work, we focus on the Java Virtual Machine Bytecode [13] and propose an end-to-end solution for dealing with what is referred to in an SPL extractive approach as *Feature identification*. Feature identification consists in analyzing and comparing the existing product variants to identify commonality and variability in terms of features.

Instead of considering the source code of the product variants, we analyze the bytecode to identify features. This approach allows a generic end-to-end solution that can: i) Support any programming language based on the bytecode (e.g., Java, Scala, Groovy, Kotlin, etc.); ii) Perform feature identification on legacy systems where the source code is not available. We implemented our solution inside the BUT4Reuse tool [14].

This paper thus makes the following contributions: i) an original approach for commonality and variability identification between the bytecode of a set of product variants; ii) a full integration of the approach with an existing framework for SPL extraction, BUT4Reuse. This solution reuses a set of techniques for feature identification and location, reusable asset construction, variability model synthesis, and visualization; iii) an assessment of the proposed approach with one real case study.

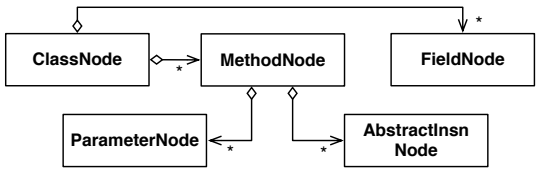

Fig. 2. ASM Model

The paper is structured as follows: Section II presents the background concepts and states our motivations. Section III details the technical workflow of our solution. Section IV reports on the evaluation of the solution with an industrial case study. Section V reports on related work. Finally, Section VI sketches the perspectives to this work.

II. Background and Motivations

A. Software Product Line Extraction with BUT4Reuse

In previous work, we introduced BUT4Reuse (Bottum-Up-Technologies for Reuse) [15], [16], a generic and extensible framework for extractive SPL adoption. It is generic by enabling its use in different scenarios with product variants of different software artifact types (e.g., source code in Java, C, models, requirements, or plugin-based architectures). It is extensible by combining different concrete techniques for the relevant activities of extractive SPL adoption (i.e., feature identification, feature location, mining feature constraints, extraction of reusable assets, feature model synthesis and visualizations). Several validation studies of BUT4Reuse using different software artifact types or different extensions have already been published [12], [17].

BUT4Reuse relies on *adapters* to support different artifact types. These adapters are the main extensible, pluggable components of the framework. An adapter is responsible for decomposing each artifact type into the constituting, distinct and atomic *elements*. It also defines how a set of elements should be constructed to create a reusable asset. Designing an adapter for a given artifact type requires three main tasks. The first task is to **identify the *elements*** that compose an artifact, at a selected granularity level. For the same artifact type, we can select from coarse to fine granularity (e.g., package level versus statement level for source code). The second task is to **define a similarity metric** between any pair of elements. An element comparing its similarity with another element returns as output a value ranging from zero (completely different) to one (identical). The last task is to **identify structural dependencies** for the elements. When the artifact type is structured, the elements will have containment relations. In the case of source code, program dependence graphs usually capture this information.

B. Bytecode as a universal JVM representation

The Java Virtual Machine (JVM) [13] is a widely spread implementation of abstract computing machines [18], originally designed to interpret programs written in Java. Its defining feature as a platform (hardware / operating system)-independent execution environment comes from its instruction set, the Java Bytecode. Programs in several popular programming languages can compile to the Java Bytecode (e.g., Java, Kotlin, Scala, etc.).

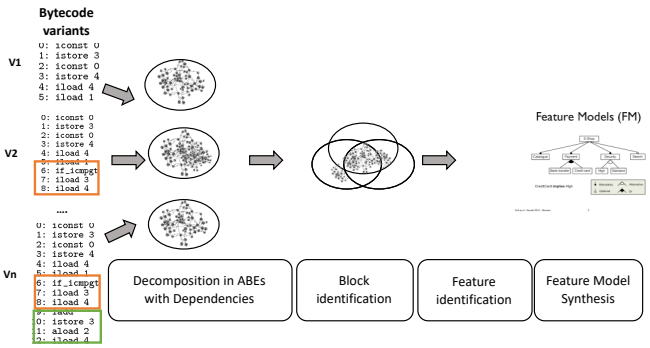

Fig. 3. Our approach

Many libraries, such as ASM [19], provide an abstraction for manipulating the bytecode. ASM allows to handle the elementary instructions described above from a high-level model. Figure 2 illustrates a partial view of this model. It follows an object orientation, presenting concepts such as `ClassNode` that abstracts the bytecode to represent a class. Each `ClassNode` is composed of `MethodNode` and `FieldNode`, that abstract, respectively, methods and fields. The `AbstractInsNode` element abstracts the bytecode related to the statements inside a method.

III. Software Product Line Extraction from Bytecode variants

Figure 3 illustrates our approach, which runs in four steps.

a) ***Decomposition in Atomic Bytecode Elements and identification of dependencies****:* This step takes as input the bytecode of a collection of system variants created using the Clone-And-Own approach. It decomposes each variant as a set of Atomic Bytecode Elements (ABEs). ABEs correspond to the main elements in the ASM model partially presented in Fig. 2. In addition, dependencies between elements are inferred at this stage. To implement this step, our approach uses the different visitors proposed by ASM to go through the bytecode and identify the ABEs.

b) ***Block Identification****:* We reuse our algorithm, called *Interdependent Elements* [20], which automatically identifies sets of ABEs that correspond to the distinguishable features from the model variants. We named *Blocks* these sets of ABEs. The detailed description of this algorithm was presented in [20] and [12]. We only underline in the following the important principles. The algorithm computes the interdependence among ABEs. Given a set of bytecode variants BVs, two ABEs abe_1 and abe_2 (of models from BVs) are interdependent if the two following conditions are fulfilled:

1) $\exists B \in BVs \;\; abe_1 \in B \wedge abe_2 \in B$;
2) $\forall B \in BVs \;\; abe_1 \in B \Leftrightarrow abe_2 \in B$.

A *Block* is thus a set of interdependent ABEs that are distinguishable in the bytecode variants.

During this step, we also extract *requires* and *exclusion-mutual* dependencies between blocks based on the ABEs dependencies. This extraction is defined as follows:

1) a block $B1$ ***requires*** another block $B2$ iff $\exists abe_1 \in B1$ and $\exists abe_2 \in B2$ and abe_1 ***requires*** abe_2;
2) a block $B1$ is in ***mutual-exclusion*** dependency with and block $B2$ iff $\exists abe_1 \in B1$ and $\exists abe_2 \in B2$ and abe_1 is in ***mutual-exclusion*** dependency with abe_2.

*c) **Feature Identification**:* This step is a semi-automatic process where domain experts manually review the elements from the identified blocks to map them with the functionalities (i.e., features) of the system. BUT4Reuse integrates *VariCloud* [21], a technique that analyzes the elements inside each block and extracts words that help domain expert to identify features. VariCloud uses information retrieval techniques, such as the IF-TDF (frequency-inverse document frequency), to analyze the text describing elements inside blocks. The descriptions used by BUT4Reuse to build word clouds are thus provided by the specific adapter. For our bytecode adapter, words correspond to the names of classes, fields, and methods.

*d) **Feature Model Synthesis**:* Once features are identified, the last step is to synthesize the feature model. Feature model synthesis represents itself an independent research field [22]. BUT4Reuse integrated many simple techniques such as the *alternatives before hierarchy*. This technique is based on calculating first the alternative constructions from the *mutual exclusion* constraints, and then creating the hierarchy using the *requires* constraints. Constraints not included in the hierarchy are added as cross-tree constraints.

IV. Evaluation

ArgoUML is an open source tool for modeling using UML diagrams. Variants for this tool originate from its Java codebase by removing specific features [23]. These features are mainly related to the tool support for the edition of different kinds of UML diagrams. We considered the bytecode of the original ArgoUML and the seven variants related to diagram edition support (8 artifacts in total). Table I shows these variants along with their the number of lines and atomic bytecode elements. As an example, the first row, `ActivityDisabled`, means that this tool variant contains all the features related to UML diagrams edition except for Activity diagrams.

Figure 4 shows the distribution of the identified blocks (colored stripes) over the artifacts (vertical bars) of the eight variants of ArgoUML. Block 0 (first stripe in all variants) is the most common one, representing the core of the ArgoUML product line. Figure 5 shows the distribution of the identified blocks over the eight ArgoUML variants. Again Block 0 is present in all variants, while Block 4 is present in all variants except `CollabDisabled`, which corresponds to the ArgoUML variants without the support of Collaboration diagrams.

TABLE I
ArgoUML variants: number of Lines of code and instructions in the bytecode.

	LOC	ABEs
ActivityDisabled	118,066	24282
CollabDisabled	118,769	26085
DeployDisabled	117,201	25912
Loggingdisabled	118,189	26010
Original	120,348	26289
SequenceDisabled	114,969	25570
StateDisabled	116,431	25687
UsecaseDisabled	117,636	25906

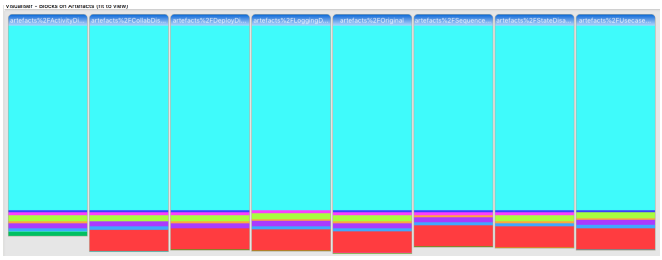

Fig. 4. Bars represent the artifacts of the eight variants, stripes the distribution of blocks over the artifacts

Figure 6 shows the VariClouds for Block 3 (above the black line) and Block 4 (below the black line). Their contents reveal that Block 3 (fourth stripe in Fig. 4) implements the support of Sequence diagrams, while Block 4 (fifth stripe in Fig. 4) implements the support of Collaboration diagrams. This confirms why Block 4 is missing in the `CollabDisabled` variant, as shown in Fig. 5.

Table II shows the identified blocks and their respective sizes in terms of ABEs. ArgoUML variants' specific implementing features can be easily identified in the first seven blocks, by manually inspecting their corresponding VariClouds, as illustrated in Fig. 6. The other remaining blocks contain fewer ABEs. After analysis of these remaining blocks, we identified them as blocks implementing the concept named *feature interaction*. Indeed, to implement any pair of two distinct features, it is common to add some implementation elements to glue these features.

The outcomes of our experiment are available online[1], to allow the reader to reproduce the same experiment.

V. Related Work

The JVM Bytecode has been subject to an important body of research investigations for a wide range of purposes, such as bug, vulnerability, malware detection, program verification and automatic testing, code obfuscation, and so on. The closest research line in our context studies code clone detection. Code clones may share a syntactic or functional similarity. Syntactic similarity arises with similar code patterns. There are three types of clones in this category. Type-1 clones are exact copies, with perhaps differences in whitespace, comments, or layout. In Type-2 clones, variables or functions have been renamed,

[1] https://pages.lip6.fr/Tewfik.Ziadi/iceccs18.html

	Block 00	Block 01	Block 02	Block 03	Block 04	Block 05	Block 06	Block 07	Block 08	Block 09	Block 10	Block 11	Block 12	Block 13	Block 14	Block 15	Block 16	Block 17	Block 18	Block 19	Block 20	Block 21
ActivityDisabled	X	X	X	X	X	X	X		X	X	X	X	X	X							X	
CollabDisabled	X	X	X	X		X	X	X		X		X	X	X	X	X	X	X	X			
DeployDisabled	X	X	X	X	X	X		X	X	X	X		X	X	X	X	X		X	X		
LoggingDisabled	X		X	X	X	X	X	X	X							X	X	X	X	X		X
Original	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
SequenceDisabled	X	X	X		X	X	X	X			X	X	X	X	X		X	X	X	X		
StateDisabled	X	X	X	X	X		X	X	X	X	X	X		X	X	X		X	X	X		
UsecaseDisabled	X	X		X	X	X	X	X	X	X	X	X	X		X	X	X	X		X		

Fig. 5. Distribution of identified blocks over the ArgoUML variants

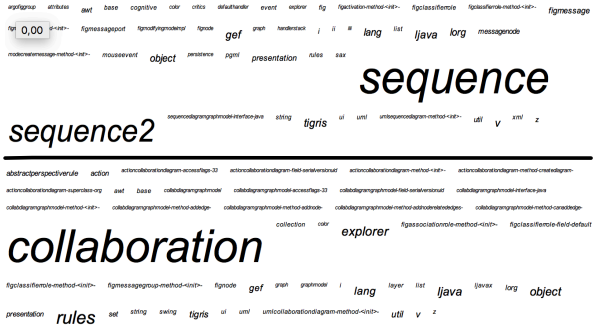

Fig. 6. Words extracted for Blocks 3 and Block 4

TABLE II
NUMBER OF ATOMIC BYTECODE ELEMENTS PER BLOCK AND IDENTIFIED FEATURES IN THE ARGOUML CASE STUDY

	ABEs	Identified feature
Block 0	21310	Core
Block 1	243	Cognitive
Block 2	376	Use cases
Block 3	688	Sequence
Block 4	195	Collaboration
Block 5	588	State
Block 6	370	Deployment
Block 7	2440	Activity

or types have changed. Type-3 clones have added or deleted statements. Type-3 clones are also termed *gapped clones* [24]. Functional similarity refers to semantic similarity in Type-4 clones, regardless of syntactic code patterns.

Approaches in code clone detection generally fall into six categories according to the underlying technique: text-based [25], token-based [26], tree-based [27], graph-based [28], metrics-based [29], and hybrid [30]. Some code clone detection techniques have considered the bytecode as the primary artifact [31]–[33], to detect Type 3 and 4 clones.

The Clown-and-Own paradigm in SPLE yields clones that span across the four types. In `Type-1`, product variants may share the same code, except for whitespace, comments, and layout. This type of clones is rare in the context of meaningful SPLE. In `Type-2`, some product variants might have renamed, even changed the types of some variables and functions. In Type-3, some product variants might have enhanced, updated, added, or removed some features of the product line according to the products development strategy and customers' needs. Finally, in `Type-4`, some algorithms might have been redesigned (e.g., changing or tuning a sorting algorithm for performance reasons), or the developers might have switched to other programming languages to craft the same features.

We would classify our solution as targeting all types of clones. However, it bears some notable differences with other code clone detection approaches. Firstly, it is semi-automated, and each instance of extraction is domain-specific: the developer is always in the loop since s/he writes the corresponding adapter in the BUT4Reuse framework. Secondly, our solution is more generic and flexible, as it allows the product line engineer to adapt the extraction procedure to the semantics of similarity s/he considers. We believe this flexibility is relevant in many industrial contexts. We experimented an extraction approach where statements in methods' bodies were considered (not reported in this paper) and another where they were discarded. In terms of granularity in extracting feature-implementing blocks, we consider methods and classes (thus identified thanks to their signature) as first-class units of decomposition. Finally, there is no single tool that can comprehensively handle all four types of clones, the costs of complexity and scalability just being too high. Hence the flexibility of our approach makes it amenable to business context-focused applications, without paying a costly upfront investment.

VI. CONCLUSION

The extraction of product lines from software artifacts has demonstrated its relevance in leveraging a more systematic, model-based product-line approach in developing large families of related software. It can help increase productivity in companies and Open Source communities who produce large code bases using a clown-and-own approach. The main trend of the extraction paradigm is to consider source code. This paper presented an extraction approach from the Java Virtual Machine (JVM) Bytecode, to handle legacy software systems for which the source code is unavailable, and factor in all input programming languages for the JVM. An early experiment was carried out on the ArgoUML case study and preliminary results discussed.

The first perspective to this work is to consider the behavior of methods, as mentioned in the previous section. To this aim, we will adapt traditional clone detection techniques that deal with programs semantics. The second perspective is to have our adapter propose different options regarding the flexibility

of the bytecode decomposition approach, related to different granularities of extraction. Last, we will enrich the evaluation with various product families implemented in several JVM-based languages.

ACKNOWLEDGEMENT

This work was partially supported by the ITEA3 15010 REVaMP2 project: FUI the Ile-de-France region and BPI in France. Special thanks to our students Koita Demba and Lazovic Sasa for their active involvement in implementing the adapter.

REFERENCES

[1] F. van der Linden, K. Schmid, and E. Rommes, *Software product lines in action - the best industrial practice in product line engineering*. Springer, 2007.

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

[3] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Carnegie-Mellon University Software Engineering Institute, Tech. Rep., 1990.

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

[5] C. W. Krueger, "Easing the transition to software mass customization," in *Software Product-Family Engineering, 4th International Workshop, PFE 2001, Bilbao, Spain, October 3-5, 2001, Revised Papers*, ser. Lecture Notes in Computer Science, vol. 2290. Springer, 2001, pp. 282–293.

[6] C. Kästner, A. Dreiling, and K. Ostermann, "Variability mining: Consistent semi-automatic detection of product-line features," *IEEE Trans. Software Eng.*, vol. 40, no. 1, pp. 67–82, 2014.

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

[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 *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*. IEEE Computer Society, 2013, pp. 25–34.

[9] R. E. Lopez-Herrejon, T. Ziadi, J. Martinez, and A. K. Thurimella, "Second international workshop on reverse variability engineering (REVE 2014)," in *18th International Software Product Line Conference, SPLC '14, Florence, Italy, September 15-19, 2014*. ACM, 2014, p. 354.

[10] L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Variability extraction and modeling for product variants," *Software and System Modeling*, vol. 16, no. 4, pp. 1179–1199, 2017.

[11] T. Ziadi, C. Henard, M. Papadakis, M. Ziane, and Y. L. Traon, "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*, Y. Cho, S. Y. Shin, S. Kim, C. Hung, and J. Hong, Eds. ACM, 2014, pp. 1064–1071.

[12] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Automating the extraction of model-based software product lines from model variants (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, M. B. Cohen, L. Grunske, and M. Whalen, Eds. IEEE Computer Society, 2015, pp. 396–406.

[13] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java® Virtual Machine Specification, Java SE 8 Edition*, ORACLE, https://docs.oracle.com/javase/specs/jvms/se8/html/.

[14] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Bottom-up technologies for reuse: automated extractive adoption of software product lines," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, S. Uchitel, A. Orso, and M. P. Robillard, Eds. IEEE Computer Society, 2017, pp. 67–70.

[15] ——, "Bottom-up adoption of software product lines: a generic and extensible approach," in *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*. ACM, 2015, pp. 101–110.

[16] ——, "Bottom-up technologies for reuse: automated extractive adoption of software product lines," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, S. Uchitel, A. Orso, and M. P. Robillard, Eds. IEEE Computer Society, 2017, pp. 67–70.

[17] L. Li, J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Mining families of Android applications for extractive SPL adoption," in *Proceedings of the 20th International Conference on Software Product Lines, SPLC 2016, Beijing, China, September 19-23*, 2016.

[18] S. Diehl, P. H. Hartel, and P. Sestoft, "Abstract machines for programming language implementation," *Future Generation Comp. Syst.*, vol. 16, no. 7, pp. 739–751, 2000.

[19] E. Bruneton, R. Lenglet, and T. Coupaye, "Asm: A code manipulation tool to implement adaptable systems," in *In Adaptable and extensible component systems*, 2002.

[20] T. Ziadi, L. Frias, M. A. A. da Silva, and M. Ziane, "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*. IEEE Computer Society, 2012, pp. 417–422.

[21] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Name suggestions during feature identification: The VariClouds approach," in *Proceedings of the 20th International Conference on Software Product Lines, SPLC 2016, Beijing, China, September 19-23*, 2016.

[22] S. She, U. Ryssel, N. Andersen, A. Wasowski, and K. Czarnecki, "Efficient synthesis of feature models," *Information and Software Technology*, vol. 56, no. 9, 2014.

[23] M. V. Couto, M. T. Valente, and E. Figueiredo, "Extracting software product lines: A case study using conditional compilation," in *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, ser. CSMR '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 191–200.

[24] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, "On detection of gapped code clones using gap locations," in *APSEC*. IEEE Computer Society, 2002, pp. 327–336.

[25] J. Svajlenko and C. K. Roy, "Fast and flexible large-scale clone detection with cloneworks," in *ICSE (Companion Volume)*. IEEE Computer Society, 2017, pp. 27–30.

[26] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy, "CCAligner: a token based large-gap clone detector," in *Proceedings of The 40th International Conference on Software Engineering (ICSE 2018)*, 21018.

[27] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *ASE*. ACM, 2016, pp. 87–98.

[28] Y. Higo, Y. Ueda, M. Nishino, and S. Kusumoto, "Incremental code clone detection: A pdg-based approach," in *WCRE*. IEEE Computer Society, 2011, pp. 3–12.

[29] E. Choi, N. Yoshida, T. Ishio, K. Inoue, and T. Sano, "Extracting code clones for refactoring using combinations of clone metrics," in *IWSC*. ACM, 2011, pp. 7–13.

[30] Y. Jia, D. Binkley, M. Harman, J. Krinke, and M. Matsushita, "KClone: A Proposed Approach to Fast Precise Code Clone Detection," in *Third International Workshop on Detection of Software Clones (IWSC)*, 01 2009.

[31] D. Yu, J. Wang, Q. Wu, J. Yang, J. Wang, W. Yang, and W. Yan, "Detecting java code clones with multi-granularities based on bytecode," in *COMPSAC (1)*. IEEE Computer Society, 2017, pp. 317–326.

[32] I. Keivanloo, C. K. Roy, and J. Rilling, "Sebyte: Scalable clone and similarity search for bytecode," *Sci. Comput. Program.*, vol. 95, pp. 426–444, 2014.

[33] A. Cuomo, A. Santone, and U. Villano, "Cd-form: A clone detector based on formal methods," *Sci. Comput. Program.*, vol. 95, pp. 390–405, 2014.

[34] S. Uchitel, A. Orso, and M. P. Robillard, Eds., *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*. IEEE Computer Society, 2017.