Bridging the Gap Between Clone-and-Own and Software Product Lines

Timo Kehrer
Humboldt University of Berlin, Germany
timo.kehrer@informatik.hu-berlin.de

Thomas Thüm
University of Ulm, Germany
thomas.thuem@uni-ulm.de

Alexander Schultheiß
Humboldt University of Berlin, Germany
alexander.schultheiss@informatik.hu-berlin.de

Paul Maximilian Bittner
University of Ulm, Germany
paul.bittner@uni-ulm.de

***Abstract*—Software is often released in multiple variants to meet all customer requirements. While *software product lines* address this need by advocating the development of an integrated software platform, practitioners frequently rely on ad-hoc reuse based on a principle which is known as *clone-and-own*. This practice avoids high up-front investments, as new variants of a software family are created by simply copying and adapting an existing variant, but maintenance costs explode once a critical number of variants is reached. With our research project VARIANTSYNC, we aim to bridge the gap between clone-and-own and product lines by combining the minimal overhead and flexibility of clone-and-own with the systematic handling of variability in software product lines. The key idea is to *transparently* integrate product-line concepts with variant management facilities known from *version control systems* in order to automatically synchronize a set of evolving variants. We believe that VARIANTSYNC has the potential to change the way how practitioners develop multi-variant software systems for which it is hard to foresee which variants will be added in the future.**

***Index Terms*—Software product lines, clone-and-own, feature traceability, configuration management**

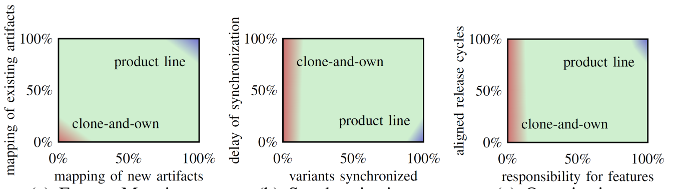
I. Introduction

Context and Motivation — The need for software mass-customization has been recognized a long time ago within Parnas' research on program families in the 1970s [1]. The field later evolved into *software product-line engineering* (SPLE) [2], [3], promoting techniques to systematically manage software variability. A *software product line* (SPL) is a set of similar software products (i.e., variants) with well-defined commonalities and variabilities, developed based on an integrated software platform. Each product of an SPL is identified by a unique combination of features (aka. configuration) [3], a feature being an end-user visible characteristic of a product. The set of valid feature combinations (i.e., configurations) is typically specified in a feature model [4], [5]. A product line is implemented by mapping the features onto implementation artifacts and choosing a variation mechanism which specifies how to generate individual products (re-)using the common artifacts (e.g., using preprocessors, build systems, or plug-ins) [3], [4], [6]. Ideally, deriving a variant then amounts to selecting the desired features in a configurator tool.

However, product lines have high up-front investments, require a radical change in the development organization, and the actual return-on-investment is often hard to estimate [7], [8], [9]. Since it is typically unknown which and how many variants are required in the future [10], [11], the state-of-practice in engineering multi-variant software systems often follows a common pattern: The development starts with a single variant, where the overhead of product lines does not pay off. Later, further variants are added by copying and adapting an existing variant, and all of these forked variants may evolve in parallel; a principle which is generally known as clone-and-own [7], [8], [10], [12]. However, if a critical number of variants is reached, maintaining such a software family becomes impractical. Propagating changes such as bug fixes to other variants is increasingly difficult and ambiguous for a growing number of variants [7], [8], [10], [12], [13].

Research Vision and Approach — With our research project VARIANTSYNC, we aim to overcome this dilemma through a new development methodology which bridges the gap between clone-and-own and product lines. The goal is to get rid of the limitations of both approaches while preserving their advantages. We want to combine and integrate the minimal overhead of clone-and-own with the systematic handling of variability of SPLs. Instead of forcing developers to employ a heavyweight product-line process, we *transparently support their on-going development with concepts and techniques from software product-line engineering and version control systems.* In our envisioned methodology, software development is performed according to a session-oriented editing model where the features being touched during an editing session are made explicit by developers. This way, software artifacts of all variants shall be mapped to features, drawn from a common set of features, with minimal input by developers. The gathered domain knowledge shall be exploited to support the synchronization of variants during evolution. That is, improvements in one variant shall be propagated to all other variants sharing the affected features in a highly automated fashion. Finally, the domain knowledge shall also serve as a basis for automatically creating variants that implement a new feature combination.

© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.



(a) Feature Mapping (b) Synchronization (c) Organization

Fig. 1. Motivation, research vision, and practical impact of VariantSync.

Practical Impact — Fig. 1 illustrates how VariantSync (green area) bridges the gap between clone-and-own (red spot or area) and product lines (blue spot) w.r.t. three dimensions, emphasizing the flexibility and novel characteristics of our proposed methodology. Fig. 1(a) focuses on the degree of completeness of feature mappings. Whereas each artifact of a product line is mapped to features [3], [4], there is no such mapping for clone-and-own [14]. With VariantSync, any subset of both existing and newly developed artifacts may be mapped to features. Fig. 1(b) shows the extent to which artifacts implementing the same feature are synchronized among variants, and the delay of such a synchronization. As artifacts are reused across variants in product lines [3], [4], all possible variants are instantly synchronized. With clone-and-own, changes to artifacts in one variant are propagated to other dedicated variants on demand [12], [15], typically leading to very few variants being synchronized. VariantSync enables the synchronization of changes into any subset of variants at any time. Fig. 1(c) addresses the organization of the development of a family of software variants. With product lines, release cycles of variants need to be identical and developers are responsible for features. In clone-and-own, release cycles of variants are often not aligned [10], [11] and developers are typically responsible for variants [7], [8], [10], [12], [13] rather than features. VariantSync enables arbitrary release cycles of individual variants and supports the concurrent development of both features and variants.

II. State-of-the-Art and Research Gaps

Effectively managing the evolution of variant-rich software systems is still among the main challenges of software engineering [16], [17], though being tackled from many different angles. The terminology introduced by Antkiewicz et al. [10] distinguishes seven levels in the continuum between ad-hoc clone-and-own (L0) and rigorous SPLE (L6). While product derivation is fully automated on level L6, it may be followed by a manual post-processing on level L5. This adds some flexibility, but leads to the problem of how to consistently co-evolve individual products and centrally managed development artifacts [18]. At the other side of the spectrum, level L1 enhances ad-hoc clone-and-own by collecting *provenance data* about variants, such as revision graphs created through branching capabilities of version control systems [19], potentially extended by annotating commits with variability information [20]. The management of parallel development branches is supported by basic repository services. However, classical merging [21] is inadequate for synchronizing variants. Variants are supposed to *have differences*, whereas merging tries to get rid of them by integrating all the changes into a unified version. The synchronization of variants can only be achieved by propagating the desired changes from one branch to another, which is typically referred to as *cherry-picking changes* [22] and which is still a labor-intensive task prone to errors [23]. In the sequel, we will have a closer look at related work which, like our approach, aims to improve the development according to levels L2, L3, and L4, where clone-and-own is further supported by exploiting additional domain knowledge such as features, configurations, and feature models.

Some approaches aim to support clone-and-own by keeping the traditional version space organization of version control systems. Rubin et al. [7], [24], present a set of conceptual operators which, in addition to deriving new variants, can be also used to propagate features between variants. However, they do not provide concrete instantiations of the proposed operators, but only discuss the applicability of existing techniques. By simulating the development of a family of cloned projects, Ji et al. [14] evaluated the potential of embedded feature annotations to propagate changes between these projects. However, the simulation study has been conducted in a purely manual fashion. Solutions of how feature mappings can be efficiently gathered and maintained as well as how features can be propagated in an automated fashion are out of the scope of their study. In sum, while existing research indicates the potential benefits of entering the area between ad-hoc clone-and-own and software product lines (Fig. 1), to date, this would require extensive manual effort and is not yet feasible.

Dating back to ideas of Conradi and Westfechtel [19], *variation control systems* [25], [26], [27] and *filtered SPLs* [28] get rid of parallel development branches by using an integrated software platform. As opposed to classical SPLE, developers work on a single variant which is a *projection* of the SPL and which is obtained from and re-integrated back into the integrated platform using the established *check-out/modify/commit workflow* of version control systems. Linsbauer et al. use their tool ECCO and combinatorics of configurations to map features to parts of artifacts and to semi-automatically extract, compose, and complete new variants [9], [29]. However, alternative organizations of the version space never made it into mainstream version control systems [30]. Methodologies based on an integrated software platform suffer from similar organizational constraints and flexibility problems as traditional SPLs (cf. Fig. 1), and there is no control on how changes in one variant affect other variants.

Finally, there is a whole line of research which aims to support the migration of a set of variants into an SPL. Thereby, feature locations across the existing implementation artifacts need to be *recovered* after the fact [9], [29], [31], [32], [33], [34]. All works on feature location recovery lament the lack of precision and the high effort of the task, even if using a semi-automated technique [14], [35]. A migration requires to *stop the development* of all variants, sometimes for months or years [7], [32]. The migration to a product line may even fail and thus bears considerable *economical risks* [34].

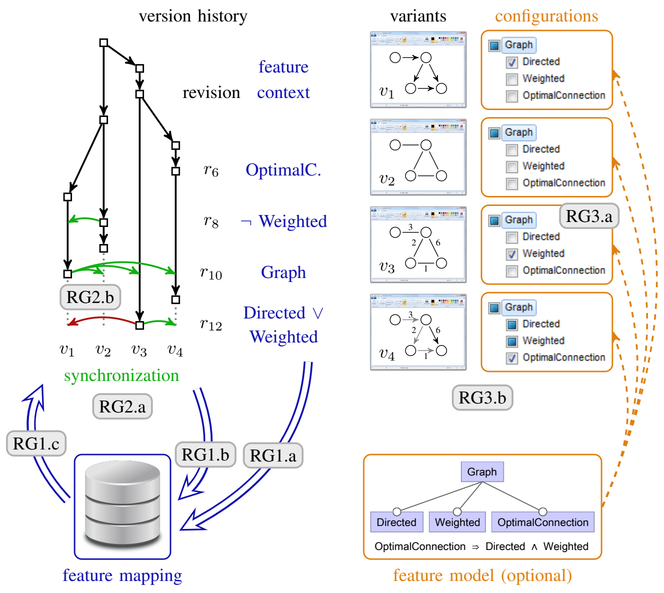


Fig. 2. Illustration of the VARIANTSYNC approach and research goals.

III. Overview of the Approach

Basic Assumptions — To gather domain knowledge on features and configurations, we rely on two reasonable assumptions which are justified within in the literature. First, we assume that developers agree on a common nomenclature for the domain (i.e., features) and, for a given variant, the respective developers know which subset of features is implemented by that variant [9], [11], [29]. The set of available features is typically specified by the product documentation [36], and can also be retrieved from other departments within an organization (e.g., the sales department knows about product features) [9]. Second, we assume that developers know which feature or feature interaction they are working on [14]. Kästner et al. [32] argue that developers typically have this piece of domain knowledge, and expert interviews on industrial clone-and-own show that it is most pronounced during implementation [8].

Development Methodology — Our development methodology follows a session-oriented editing model, in which features being touched during an editing session are made explicit by developers in terms of a propositional formula over features, referred to as *feature context* of that editing session. This is illustrated in Fig. 2, using a snapshot of the development of a simple graph management software. The snapshot comprises four features and four variants implementing different configurations (upper right). The historical evolution of these variants is shown by the revision graph in the upper left, annotated by the feature contexts which led to the revisions r_6, r_8, r_{10}, and r_{12}. Note that revisions in the revision graph shown in Fig. 2 encapsulate editing sessions associated to a dedicated feature context. They may be committed to a version control system to become repository revisions, but editing sessions may also be more fine-grained than repository commits.

Feature contexts and configurations can be used to identify the target variants which need to be synchronized in response to evolution steps. For example, no synchronization is needed in response to the evolution step leading to revision r_6, as it affects the feature *OptimalConnection* which is only comprised by the modified variant v_4. In contrast, the changes leading to revision r_8 of variant v_2 have been performed under feature context $\neg$*Weighted* and thus shall be propagated (green arrow) to variant v_2, which does not comprise that feature. Similarly, the changes to the shared feature *Graph* in the evolution step leading to revision r_{10} of variant v_1 shall be propagated to all other variants. The change resulting in revision r_{12} of variant v_3 needs to be propagated to two variants. The red arrow symbolizes that the synchronization of variant v_1 cannot be performed automatically due to synchronization conflicts.

The feature context is not only used to determine target variants, but also to establish a mapping from features to artifacts in the currently edited variant as well as in other variants (bottom left). This feature mapping may also be employed to derive new variants and to reconfigure existing variants. For example, we may obtain a new variant comprising the features *Directed* and *Weighted* by starting from variant v_1 and transferring an implementation of feature *Weighted* from variant v_3 or v_4, or by starting from variant v_4 and deleting all parts related to feature *OptimalConnection*. In general, an additive strategy assumes that the start variant comprises a subset of the features of the desired variant. For each feature which is to be added, we must choose a suitable *donor* variant for obtaining its implementation. A subtractive reconfiguration strategy assumes that the set of features contained by the start variant is a superset of the features which shall be comprised by the desired variant. Those parts implementing the superfluous features must be removed.

IV. Research Goals

Realizing our vision is faced with fundamental technical challenges from which we derive the following research goals:

RG1: *Transparent Collection and Management of Feature Mappings*: To physically propagate artifact changes, a basic prerequisite is that features are mapped to development artifacts or fragments thereof. Our goal is to reach a high degree of feature mapping with minimal manual effort. First, we aim at automated techniques to establish a feature mapping already during development (RG1.a in Fig. 2). Second, we are going to synchronize feature mappings along with the synchronization of artifact changes when variants evolve (RG1.b). Finally, to support the completion and correction of feature mappings, we even want to synchronize mappings between variants without any associated artifact changes (RG1.c).

RG2*: Propagation of Changes Between Variants*: By collecting knowledge about which features are relevant for which variants and which features are involved in which changes, we want to automate the synchronization of variants by propagating changes between them. Thereby, we aim at supporting development teams for variants, as in clone-and-own, and development teams for features, as in SPLE (RG2.a). For both scenarios, synchronization conflicts and other kinds of change propagation failures shall be handled with minimal developer interaction for a set of synchronization targets (RG2.b).

RG3: *Reconfiguration of Existing and Generation of New Variants*: Even if synchronizing a set of variants is fully automated, developers would still be challenged by the creation of new variants and when features need to be added to or removed from existing variants. Hence, we intend to enable the *reconfiguration of existing* (RG3.a) and the *creation of new variants* (RG3.b) by means of feature selections. For both scenarios, our goal is to provide maximal confidence that a synthesized variant actually behaves as expected, and to minimize developer interactions during variant generation.

V. Early Results and Validation

Research Prototype — While we aim at developing a conceptual framework which abstracts from technical details and tooling issues as far as possible, a concrete research prototype is required for evaluating our approach. To that end, we have developed a first version of such a prototype as an Eclipse plug-in which works on a textual representation of source code artifacts [37]. For the sake of rapid prototyping, artifact changes and feature mappings are treated in a line-based manner. The main purpose of the research prototype in its current state is to show the general feasibility of our approach.

Refinement of Technical Challenges — Our research prototype also helped us to better understand and refine the technical challenges imposed by our research goals, leading us to the consequent steps to reach our overall vision.

As for RG1, it became obvious that an automated recording of feature mappings needs at least basic syntactic information of the underlying development documents in order to prevent ill-formed mappings. Inspired by the concept of disciplined annotations [38], [39], features shall be mapped to nodes of an Abstract Syntax Tree (AST) instead of source code lines that do not have to follow a certain structure.

As for RG2, to automatically identify synchronization targets, we will use product-line analysis techniques [40], [41], [42] for exploiting the feature contexts of changes as well as the knowledge which feature selection is implemented by which variant. However, the actual propagation of changes between variants may fail due to synchronization conflicts and needs to be handled by developers. Our goal is to minimize developer interactions by lifting the handling of synchronization conflicts to the level of entire software families instead of relying on laborious pairwise conflict resolutions as practiced today. In particular, synchronization conflict resolutions shall be treated as reusable assets which can be transferred to all target variants affected by the same conflict.

Concerning RG3, as illustrated in Sec. III, the reconfiguration of existing and generation of new variants may be performed in several ways. Our hypothesis is that optimal solutions may be only obtained by combining the basic strategies (i.e., additive and subtractive) for a given reconfiguration case. The synthesized result depends on the selected start variant (when generating new variants) and on the selected donor variants (in case of an additive approach). This renders reconfiguration into a combinatorial optimization problem which we aim to tackle by using meta-heuristic algorithms as known from search-based software engineering [43].

Evaluation Plan — The main evaluation goal is to assess to which extent our approach enables to bridge the gap between clone-and-own and software product lines, as illustrated in Fig. 1. That is, we are interested how different factors influence the degree of automation feasible for variant synchronization. In particular, we aim to study different percentages of feature mappings for existing as well as new artifacts. Furthermore, we are going to evaluate how the number of synchronized variants and the delay of synchronization affects manual synchronization effort by developers. Finally, we will investigate whether development teams for variants, development teams for features, and even combinations thereof are feasible.

The ideal evaluation would be to apply our development methodology in an industrial context over months or years. While we envision such a technology transfer and industrial field study in the long run, an experiment is more feasible for the near future, in which students or real developers use our tool support for pre-defined tasks. A control group is supposed to develop the same functionality with pure clone-and-own or using product-line technologies, which eventually leads to a qualitative comparison of the development methodologies.

As for quantitative assessments, we will use product lines for which the history is available such that we can simulate their evolution. This is the most common form of evaluation for clone-and-own approaches [9], [32], [33], [44], [45], [46], which has the advantage that we have the ground truth for feature mappings and change synchronizations. A bias might be introduced by the absence of unintentional divergences between variants [20], [31], which we aim to mitigate by injecting such unintentional divergences in a controlled manner [47]. We will also use existing clone-and-own projects and their version history as experimental subjects [9], [14], [20], [34]. In this case, however, feature mappings need to be recovered from source code and commit messages. We aim to mitigate this potential bias by taking Marlin [12] as a subject, because the cloned variants contain preprocessor annotations which indicate feature mappings. Alternatively, we can use projects which have been migrated to an SPL by other researchers (cf. ESPLA repository [48]).

VI. Conclusion

Effectively managing the evolution of variant-rich software systems is still among the main challenges of software engineering. With our research project VariantSync, we tackle this challenge by a fundamentally new methodology. It combines the flexibility of clone-and-own with the systematic handling of variability of software product lines. The key idea is to transparently integrate the central product-line concepts of features and configurations with variant management facilities known from version control systems to automatically synchronize a set of evolving variants and to derive new variants.

Acknowledgment

The VariantSync project is funded by the German Research Foundation (grants KE 2267/1-1 and TH 2387/1-1).

References

[1] D. L. Parnas, "On the Design and Development of Program Families," *TSE*, vol. SE-2, no. 1, pp. 1–9, 1976.

[2] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer, 2005.

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

[4] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications.* ACM/Addison-Wesley, 2000.

[5] D. Batory, "Feature Models, Grammars, and Propositional Formulas," in *SPLC.* Springer, 2005, pp. 7–20.

[6] M. Svahnberg, J. van Gurp, and J. Bosch, "A Taxonomy of Variability Realization Techniques: Research Articles," *SPE*, vol. 35, no. 8, pp. 705–754, 2005.

[7] J. Rubin, K. Czarnecki, and M. Chechik, "Managing Cloned Variants: A Framework and Experience," in *SPLC.* ACM, 2013, pp. 101–110.

[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.* IEEE, 2013, pp. 25–34.

[9] L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Variability Extraction and Modeling for Product Variants," *SoSyM*, vol. 16, no. 4, pp. 1179–1199, 2017.

[10] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, S. Stănciulescu, A. Wąsowski, and I. Schaefer, "Flexible Product Line Engineering with a Virtual Platform," in *ICSE.* ACM, 2014, pp. 532–535.

[11] L. Linsbauer, S. Fischer, R. E. Lopez-Herrejon, and A. Egyed, "Using Traceability for Incremental Construction and Evolution of Software Product Portfolios," in *SST.* IEEE, 2015, pp. 57–60.

[12] S. Stănciulescu, S. Schulze, and A. Wąsowski, "Forked and Integrated Variants in an Open-Source Firmware Project," in *ICSME.* IEEE, 2015, pp. 151–160.

[13] R. Lapeña, M. Ballarin, and C. Cetina, "Towards Clone-and-Own Support: Locating Relevant Methods in Legacy Products," in *SPLC.* ACM, 2016, pp. 194–203.

[14] W. Ji, T. Berger, M. Antkiewicz, and K. Czarnecki, "Maintaining Feature Traceability with Embedded Annotations," in *SPLC.* ACM, 2015, pp. 61–70.

[15] D. Lettner and P. Grünbacher, "Using Feature Feeds to Improve Developer Awareness in Software Ecosystem Evolution," in *VaMoS.* ACM, 2015, pp. 11:11–11:18.

[16] T. Berger, M. Chechik, T. Kehrer, and M. Wimmer, "Software Evolution in Time and Space: Unifying Version and Variability Management," *Dagstuhl Reports*, vol. 9, no. 5, pp. 1–30, 2019.

[17] S. Ananieva, S. Greiner, T. Kühn, J. Krüger, L. Linsbauer, S. Grüner, T. Kehrer, H. Klare, A. Koziolek, H. Lönn, S. Krieter, C. Seidl, S. Ramesh, R. H. Reussner, and B. Westfechtel, "A conceptual model for unifying variability in space and time," in *SPLC.* ACM, 2020, pp. 15:1–15:12.

[18] S. Schulze, M. Schulze, U. Ryssel, and C. Seidl, "Aligning Coevolving Artifacts Between Software Product Lines and Products," in *VaMoS.* ACM, 2016, pp. 9–16.

[19] R. Conradi and B. Westfechtel, "Version Models for Software Configuration Management," *CSUR*, vol. 30, no. 2, pp. 232–282, 1998.

[20] T. Schmorleiz and R. Lämmel, "Similarity Management via History Annotation," in *SATToSE.* Dipartimento di Informatica Università degli Studi dell'Aquila, L'Aquila, Italy, 2014, pp. 45–48.

[21] T. Mens, "A State-of-the-Art Survey on Software Merging," *TSE*, vol. 28, no. 5, pp. 449–462, 2002.

[22] C. M. Pilato, B. Collins-Sussman, and B. W. Fitzpatrick, *Version Control with Subversion.* O'Reilly Media, Inc., 2004.

[23] T. Kehrer, U. Kelter, and G. Taentzer, "Propagation of Software Model Changes in the Context of Industrial Plant Automation," *Autom.*, vol. 62, no. 11, pp. 803–814, 2014.

[24] J. Rubin and M. Chechik, "A Framework for Managing Cloned Product Variants," in *ICSE.* IEEE, 2013, pp. 1233–1236.

[25] S. Stănciulescu, T. Berger, E. Walkingshaw, and A. Wąsowski, "Concepts, Operations, and Feasibility of a Projection-Based Variation Control System," in *ICSME.* IEEE, 2016, pp. 323–333.

[26] L. Linsbauer, A. Egyed, and R. E. Lopez-Herrejon, "A Variability Aware Configuration Management and Revision Control Platform," in *ICSE.* ACM, 2016, pp. 803–806.

[27] L. Linsbauer, T. Berger, and P. Grünbacher, "A classification of variation control systems," in *GPCE.* ACM, 2017, pp. 49–62.

[28] F. Schwägerl and B. Westfechtel, "SuperMod: Tool Support for Collaborative Filtered Model-Driven Software Product Line Engineering," in *ASE.* ACM, 2016, p. 822–827.

[29] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "The ECCO Tool: Extraction and Composition for Clone-and-Own," in *ICSE.* IEEE, 2015, pp. 665–668.

[30] J. Estublier, D. Leblang, A. v. d. Hoek, R. Conradi, G. Clemm, W. Tichy, and D. Wiborg-Weber, "Impact of Software Engineering Research on the Practice of Software Configuration Management," *TOSEM*, vol. 14, no. 4, pp. 383–430, 2005.

[31] B. Klatt, M. Küster, and K. Krogmann, "A Graph-Based Analysis Concept to Derive a Variation Point Design from Product Copies," in *REVE*, 2013, pp. 1–8.

[32] C. Kästner, A. Dreiling, and K. Ostermann, "Variability Mining: Consistent Semiautomatic Detection of Product-Line Features," *TSE*, vol. 40, no. 1, pp. 67–82, 2014.

[33] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Bottom-Up Adoption of Software Product Lines: A Generic and Extensible Approach," in *SPLC.* ACM, 2015, pp. 101–110.

[34] W. Fenske, J. Meinicke, S. Schulze, S. Schulze, and G. Saake, "Variant-Preserving Refactorings for Migrating Cloned Products to a Product Line," in *SANER.* IEEE, 2017, pp. 316–326.

[35] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature Location in Source Code: A Taxonomy and Survey," *JSEP*, vol. 25, no. 1, pp. 53–95, 2013.

[36] J. Rubin and M. Chechik, "A Survey of Feature Location Techniques," in *Domain Engineering: Product Lines, Languages, and Conceptual Models*, I. Reinhartz-Berger, A. Sturm, T. Clark, S. Cohen, and J. Bettin, Eds. Springer, 2013, pp. 29–58.

[37] T. Pfofe, T. Thüm, S. Schulze, W. Fenske, and I. Schaefer, "Synchronizing Software Variants with VariantSync," in *SPLC.* ACM, 2016, pp. 329–332.

[38] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory, "Guaranteeing Syntactic Correctness for All Product Line Variants: A Language-Independent Approach," in *TOOLS Europe*, M. Oriol and B. Meyer, Eds. Springer, 2009, pp. 175–194.

[39] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in Software Product Lines," in *ICSE.* ACM, 2008, pp. 311–320.

[40] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A Classification and Survey of Analysis Strategies for Software Product Lines," *CSUR*, vol. 47, no. 1, pp. 6:1–6:45, 2014.

[41] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated Analysis of Feature Models 20 Years Later: A Literature Review," *Information Systems*, vol. 35, no. 6, pp. 615–708, 2010.

[42] T. Thüm, L. Teixeira, K. Schmid, E. Walkingshaw, M. Mukelabai, M. Varshosaz, G. Botterweck, I. Schaefer, and T. Kehrer, "Towards Efficient Analysis of Variation in Time and Space," in *VariVolution.* ACM, 2019, pp. 57–64.

[43] M. Harman and B. F. Jones, "Search-based Software Engineering," *Inf. Softw. Technol.*, vol. 43, no. 14, pp. 833–839, 2001.

[44] J. Martinez, T. Ziadi, M. Papadakis, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Feature Location Benchmark for Software Families Using Eclipse Community Releases," in *ICSR.* Springer, 2016, pp. 267–283.

[45] A. Schlie, D. Wille, S. Schulze, L. Cleophas, and I. Schaefer, "Detecting Variability in MATLAB/Simulink Models: An Industry-Inspired Technique and Its Evaluation," in *SPLC.* ACM, 2017, pp. 215–224.

[46] D. Wille, T. Runge, C. Seidl, and S. Schulze, "Extractive Software Product Line Engineering Using Model-based Delta Module Generation," in *VaMoS.* ACM, 2017, pp. 36–43.

[47] A. Schultheiß, P. M. Bittner, T. Kehrer, and T. Thüm, "On the Use of Product-Line Variants as Experimental Subjects for Clone-and-Own Research: A Case Study," in *SPLC.* ACM, 2020.

[48] J. Martinez, W. K. G. Assunção, and T. Ziadi, "ESPLA: A Catalog of Extractive SPL Adoption Case Studies," in *SPLC.* ACM, 2017, pp. 38–41.