On the Use of Product-Line Variants as Experimental Subjects for Clone-and-Own Research: A Case Study

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

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

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

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

ABSTRACT

Software is often released in multiple variants to address the needs of different customers or application scenarios. One frequent approach to create new variants is clone-and-own, whose systematic support has gained considerable research interest in the last decade. However, only few techniques have been evaluated in a realistic setting, due to a substantial lack of publicly available clone-and-own projects which could be used as experimental subjects. Instead, many studies use variants generated from software product lines for their evaluation. Unfortunately, the results might be biased, because variants generated from a single code base lack unintentional divergences that would have been introduced by clone-and-own. In this paper, we report about ongoing work towards a more systematic investigation of threats to the external validity of such experimental results. Using *n*-way model matching as a representative technique for supporting clone-and-own, we assess the performance of state-of-the-art algorithms on variant sets exposing increasing degrees of divergence. We compile our observations into four hypotheses which are meant to serve as a basis for discussion and which need to be investigated in more detail in future research.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; • **General and reference → Empirical studies**; **Evaluation**.

KEYWORDS

Clone-and-Own, Experimental Evaluation, Model Matching

ACM Reference Format:
Alexander Schultheiß, Paul Maximilian Bittner, Timo Kehrer, and Thomas Thüm. 2020. On the Use of Product-Line Variants as Experimental Subjects for Clone-and-Own Research: A Case Study. In *24th ACM International Systems and Software Product Line Conference (SPLC '20), October 19–23, 2020, Montréal, QC, Canada.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3382025.3414972

SPLC '20, October 19–23, 2020, MONTREAL, QC, Canada

ACM ISBN 978-1-4503-7569-6/20/10…$15.00
https://doi.org/10.1145/3382025.3414972

1 INTRODUCTION

Today's software often needs to be released in multiple variants to meet all customer requirements. *Software product lines* [3, 9, 28] decrease development costs in the long term and can prevent variability bugs by reusing commonalities between variants in a structured manner. However, to reduce time-to-market or because the need for variability is unknown at the beginning of development, practitioners frequently rely on an ad-hoc principle known as *clone-and-own* [10, 32, 39]. New variants of a software family are created by copying and adapting an existing variant. While clone-and-own development does not require an up-front domain analysis, maintaining a family of cloned variants becomes impractical once a critical number of variants is reached [1].

Hence, exploring the continuum between software product line and clone-and-own development has gained considerable interest in research [1]. Research topics include the synchronization of cloned variants in the course of evolution [15, 32, 36], the controlled generation of new variants from existing ones [20, 32], variation control systems [23, 24], filtered product lines [30, 38], and the migration of clones to a product line [14, 25, 47].

However, most experimental evaluations of clone-and-own research use product-line variants instead of real code clones as experimental subjects [14, 24, 25, 47]. Variants generated from product lines do not expose *unintentional divergences* of semantically equivalent software fragments, which are common for cloned variants [17, 35, 36]. Therefore, experimental results might be biased and not generalizable to cloned variants that drift away from each other over time. We call the introduction of unintentional divergences *variant drift*. To the best of our knowledge, there is no systematic investigation of these threats to the external validity of experimental results obtained on product-line variants.

In this paper, we report about ongoing work towards closing this research gap. We investigate *n*-way model matching [31] as a representative technique for systematically supporting clone-and-own. *N*-way matches determine the common elements in a set of variants, and they are frequently needed as a preparatory step to migrate a set of cloned variants into a software product line [12, 16, 33, 34, 45, 46]. Based on a standard metric for assessing the quality of a matching [31], we assess the performance of *n*-way matching algorithms of varying complexity on sets of variants generated from software product lines and created through clone-and-own.[1]

[1]The most recent version of the prototype is hosted at https://github.com/AlexanderSchultheiss/variant-drift. A snapshot of the replication package can be found under https://doi.org/10.5281/zenodo.3999317

The performance is first assessed on the original variants, and we then simulate variant drift by introducing divergences through applying language-specific refactorings, observing the impact on the algorithms' performance.

In a nutshell, we found that all investigated algorithms perform better when using product-line instead of clone-and-own variants. At the same time, there are no apparent differences between the algorithms when they are evaluated on product-line variants. However, this changes drastically in the course of simulating variant drift. After injecting a few divergences into the generated product-line variants, simple algorithms are significantly outperformed by more complex ones. Interestingly, we could not observe the same effect for variants created through clone-and-own. Here, the qualitative difference between simple and complex algorithms remains rather constant in the course of injecting further divergences into the cloned variants. In summary, we contribute to the body of knowledge by

- bringing more attention to a potential threat to the validity of experimental results gathered on product-line variants,
- investigating the impact of variant drift on the empirical evaluation of techniques for n-way model matching,
- and compiling our observations into four hypotheses that can serve as a basis for further investigation and discussion.

2 N-WAY MATCHING

Detecting commonalities and differences between cloned software variants is one of the key requirements for many approaches to systematically enhancing or supporting clone-and-own development [12, 16, 31, 33, 34, 45, 46]. Detecting such commonalities and differences is typically achieved by calculating a matching among several input artifacts, commonly referred to as *n*-way matching.

Aiming at the matching of conceptual source code entities while abstracting from their textual representation, sophisticated *matchers* work on structured or semi-structured program representations [7] which can be considered as *models* of a program's abstract syntax graph. In this paper, a model is considered to be a set of *elements*, where each element comprises a set of *properties* which are plain names. For example, in an object-oriented program, elements may correspond to classes whose fields, methods and associations to other classes are represented by properties.

Intuitively, a model matching identifies groups of corresponding elements which can be considered "the same" in a subset of the set of all input models. Formally, a matching $M = \{t_1, \ldots, t_k\}$ for n input models is a set of matches $t_{i \in \{1,\ldots,k\}} = \{e_1, \ldots, e_{m_i}\}, m_i \leq n$, where each match t_i is a set containing at most one element from each distinct input model, and each element is part of exactly one match. Elements of a match should be equal or highly similar to each other. If a match comprises only a single element, this element can be considered to be unique among all input models.

For assessing the quality of a matching, we use the *weight* metric proposed by Rubin and Chechik [31], as it does not require an ideal matching as reference, which is commonly not available for clone-and-own projects. Given a match $t = \{e_1, \ldots, e_m\},\ m \leq n$, of elements from n input models, its weight is given by:

$$w(t) = \frac{\sum_{2 \leq j \leq |t|} j^2 \cdot n_j^p}{n^2 \cdot |\pi(t)|}, \tag{1}$$

where n_j^p denotes the number of properties that occur in exactly j elements of the match, and $\pi(t)$ is the set of all distinct properties in elements of the match. The idea is to assign greater weight to matches comprising more elements with a higher amount of common properties. For example, a match gets assigned the highest weight (i.e., 1), if its elements share the same set of properties, and if it comprises one element from each model. The weight of a matching (i.e., the set of matches for the input models) is then defined as the sum of weights over all matches.

We selected three matching algorithms of varying complexity in order to investigate variant drift in the context of n-way model matching. The first algorithm is a naive name-based matcher, which we refer to as *NameBased*. It matches exactly those elements of different models that have the same name. It does not consider the properties of the elements in any way. We chose this matcher as a naive baseline for the other two matchers. The second algorithm, which we refer to as *Pairwise*, performs sequential *two*-way matching of the input models using the Hungarian algorithm [19]. To be independent from input order, *Pairwise* sorts the input models descending by size. We chose this approach because matching in descending order showed the best results in an empirical evaluation of Rubin and Chechik [31]. The last algorithm, *NwM*, is a heuristic *n*-way matcher developed by Rubin and Chechik [31]. It considers all elements of all input models at the same time, which increases its chance to find the best match. A detailed description of how *NwM* proceeds can be found in the original work [31]. To the best of our knowledge, *NwM* can be considered the state-of-the-art for n-way model matching.

3 EXPLORATIVE RESEARCH METHOD

We question whether product-line variants are appropriate for the experimental evaluation of clone-and-own research. As clones are synchronized on demand [21, 40] they exhibit *unintentional divergences* that are not present in product lines. These divergences could, for example, be introduced through refactoring operations that are applied during development. Refactoring is a frequently used technique in programming that aims at improving a program's internal structure without altering its external behavior [26].

While refactoring is likely performed in both software product-line engineering and clone-and-own development, there is a significant difference in how it affects individual variants. In clone-and-own, refactoring operations are directly applied to one, some, or all of the cloned variants. This can lead to unintentional divergences, i.e., variant drift. Typically, the amount of unintentional divergences increases over time. The cloned variants which are identical at the beginning are drifting away from each other. In software product-line engineering, implementation artifacts are reused across variants [2, 8]. Hence, applied refactorings are instantly synchronised to all variants containing the refactored artifacts [22, 37].

In our controlled experiment presented in the next section, we want to investigate how increasing variant drift among product-line variants affects the quality of matchings calculated by *n*-way matchers. Therefore, we first describe the refactoring operations we use for injecting divergences in Section 3.1. In Section 3.2, we discuss how we compare matchings of model sets with different degrees of variant drift.

Table 1: Distribution of the Most Frequent Refactoring Operations Across 200 Projects Reported by Vassallo et al. [42]

Refactoring Operation	Count	Percentage
Rename Method	4, 912	29.48%
Move Field	3, 400	20.41%
Move Method	2, 031	12.19%
Extract Interface	1, 928	11.57%
Rename Class	1, 468	08.81%
Other Refactorings	2, 921	17.53%
Total	**16, 660**	**100.00%**

3.1 Injection of Unintentional Divergences

Numerous refactoring operations have been proposed in the literature, most notably the compilation of object-oriented refactorings presented by Fowler [13]. However, not all of them are applied equally often in practice. Recently, Vassallo et al. [42] analyzed refactoring practices in 200 open-source projects written in Java and found significant differences w.r.t. the frequency of their application. The five most frequent refactoring operations and their frequency of application across all 200 projects are shown in Table 1.

We derive four refactoring operations from these five most frequent refactorings, so that they can be applied to element-property models as used in this paper. Each refactoring is applied to a single model. Hence, a refactoring operation does not change multiple models simultaneously.

- *Rename Element* renames one single element and corresponds to *Rename Class*. The operation also renames all properties of all elements that contain the same name as a substring, to account for references.
- *Rename Property* changes the name of one property and corresponds to *Rename Method*.
- *Move Property* deletes a property from a source element and adds it to a target element. The operation corresponds to *Move Field/Method*. It can only be applied if the target element does not have a property with the same name.
- *Extract Interface* receives a set of elements as input that have at least one common property. The operation creates a new element with a randomly generated name and copies or moves all common properties of the input elements to this new element. The decision between copying and moving the properties to the extracted interface is made randomly, but all properties are handled in the same way for one extraction. Having two different strategies is motivated by representing both interfaces and normal classes.

3.2 Comparing Matching Quality

The weight metric defined by Rubin and Chechik [31], shown in Equation 1, can be summed up for all matches in a matching to get the matching's quality for a given set of input models. However, it should not be used to compare matchings for different sets of models, as the sum of weights is not normalized. In other words, the weight of a matching is an absolute measure which correlates with the size of the models, the similarity between elements, and the number of possible matches. Hence, we assess the quality of a matching M for a set of models $\mathscr{M}$ based on its normalized weight and the highest possible weight for a matching of the models $h(\mathscr{M})$:

$$|w(M)| := \frac{w(M)}{h(\mathscr{M})}. \tag{2}$$

A normalization of the weight $w(M)$ is necessary to make it comparable across different subjects. However, finding the required optimal solution $h(\mathscr{M})$ requires the consideration of all possible matches, and the number of possible matches for n models is $(\prod_{i=1}^{n}(k_i + 1)) - 1$, where k_i denotes the number of elements in the i-th model [31]. Thus, it is not feasible to calculate the optimal solution in a brute-force manner. Instead, we propose to calculate an upper bound for the highest possible weight of a matching.

The basic idea behind our estimation of an upper bound is that, as opposed to finding a globally optimal matching, it is relatively easy to find the best match for a specific element. To that end, for a given element, we first iterate over all models and select one element from each model that has the highest weight with respect to the given element. Then, we sort the selected elements by their weight in descending order. Lastly, we sequentially match the given element with the selected candidates. A candidate is only added to the match if this increases the weight of the match. This way, we receive the match with the highest weight for each element in the input models. The process is repeated for all elements and the final upper bound $h'(\mathscr{M})$ is then calculated as the sum over the weights of the collected matches. Note that these matches do not represent a valid matching, because a particular element can be part of several matches. Moreover, $h'(\mathscr{M})$ is an upper bound, because it considers the match with the highest weight of each element. In conclusion, we use $h(\mathscr{M}) := h'(\mathscr{M})$ in Equation 2.

4 CONTROLLED EXPERIMENT

We instantiate the general research methodology presented in the previous section in a controlled experiment. We first outline the experimental setup, and then continue by presenting our results.

4.1 Experimental Subjects

We use models of product-line variants and models of variants created through clone-and-own as experimental subjects.

The Pick and Place Unit (*PPU*) [4, 27, 43] and the Barbados Car Crash Crisis Management System (*bCMS*) [5, 6, 41] consist of variants generated from software product lines. Both were recently used by Reuling et al. [29] as a benchmark set for the empirical evaluation of their n-way model merging technique. The *PPU* is a laboratory plant comprising 16 application scenarios that differ in mechanical, electrical, and software setup of the plant. System models describing the behavior of these scenarios were presented by Vogel-Heuser et al. [44]. The *bCMS* was developed to support the distributed crisis management of accidents on public roadways. For the purpose of our evaluation, we only focus on the object-oriented implementation models of the system [6]. These models include both functional and non-functional variability of the system.

The third subject comprises variants from a software family called *Apo-Games*. The variants were developed with the clone-and-own approach [11, 32] and were recently presented by Krüger et al. as a challenge for variability mining [18]. We use models extracted from 20 Java variants presented in that challenge.

Table 2: Probabilities of Specific Refactoring Operations in Two Different Refactoring Setups

Refactoring Operation	Probability	
	Setup-A	*Setup-B*
Rename Element	36.05%	-
Rename Property	10.77%	-
Move Property	39.85%	74.94%
Extract Interface	13.33%	25.06%

4.2 Simulation of Variant Drift

For the simulation of variant drift (i.e., injection of unintentional divergences), we consider the two refactoring setups presented in Table 2. Each setup assigns probabilities to the refactoring operations presented in Section 3.1. In *Setup-A*, all four refactoring operations are applied. In *Setup-B*, only *Move Property* and *Extract Interface* are applied. We introduced *Setup-B*, where only restructuring operations are performed, in order to mitigate any bias towards the matchers that might be caused by applying rename operations. The probabilities of the refactorings are calculated based on their frequency of use in real-world software development, as empirically determined by Vassallo et al. [42] (see Section 3.1). The probabilities are normalized so that their sum is 100%.

For each setup, we inject divergences into the experimental subjects by applying increasing numbers of refactoring operations, from 0 to 400 in steps of 10 for each subject. First, for a given number of refactorings, the types of the refactorings are chosen randomly according to the probabilities in Table 2. Then, for each refactoring, a model is selected randomly, and from it a random yet suitable target within a model for applying the refactoring operation. Subsequent refactorings on the same model are applied in a non-overlapping manner to ensure that they cause a steadily increasing amount of divergences. Rename operations use randomly generated and unique names. After the given number of refactorings has been applied, the refactored models are saved and the process repeats by starting with the original, unrefactored models.

For each amount of refactorings $n \in \{0, 10, 20, \ldots, 400\}$, we generate 30 different sets of drifted variants. We repeat the same for both setups. In summary, we obtain $30 \cdot 41 \cdot 2 = 2,460$ sets of clones for each of the original experimental subjects.

4.3 Results

Figure 1 presents the average matching quality of the three model matchers presented in Section 2 on the sets of drifted models of *Setup-A*. The average matching quality for a specific number of refactorings is given as percentage of the weights' upper bound, calculated by dividing the average weight of the matchings by the average upper bound of the weight (see Section 3.2).

For matchings of models without variant drift (i.e., number of refactorings = 0), we notice differences in the achieved weights depending on the experimental subject. While all matchers achieve similar quality for unrefactored models of the *PPU*, *NwM* provides visibly better matchings for *bCMS* and *Apo-Games*. Surprisingly, *NameBased* achieves a better result on *Apo-Games* than the more complex *Pairwise*. This could indicate that *Pairwise* suffers more from the initial divergence in the clone-and-own models than *NameBased*. Another observation is that all matchers perform worse on the unrefactored models of *Apo-Games* than on the unrefactored models of *PPU* and *bCMS*. More specifically, the matchers achieve only around 6% to 6.5% of the boundary weight on *Apo-Games*, while achieving up to around 8% on *PPU* and *bCMS*. This might suggest that the clone-and-own variants of *Apo-Games* are more difficult to match than the product-line variants of *PPU* and *bCMS*.

With an increasing number of applied refactorings, two different effects become visible, based on whether the refactored models stem from product-line or clone-and-own models. For *PPU* and *bCMS*, the weights achieved by the three matchers start to diverge quickly. While the normalized weight of *NwM* increases, the weight of the most simple matcher, *NameBased*, decreases drastically. At the same time, the weight of *Pairwise* increases slightly. In contrast, there appears to be no such effect on *Apo-Games*. Here, the differences between weights of all matchers remain (mostly) constant, and the weight of all three matchers decreases slowly.

The results on the sets generated under *Setup-B* are presented in Figure 2. While the weights achieved by *NwM* and *Pairwise* are highly similar to the ones achieved on *Setup-A*, we notice a difference for *NameBased*. For the refactored models of the *PPU*, the normalized weight of *NameBased* first decreases and then increases again after around 150 refactorings. This effect can also be observed on *bCMS*, although in a weaker form. On *Apo-Games*, there is no visible difference to the results of *Setup-A*.

5 DISCUSSION

Based on the results of our first exploratory study, we formulate four hypotheses on how different kinds and degrees of unintentional divergences can affect the performance of techniques supporting clone-and-own development.

Hypothesis 1

> Techniques supporting clone-and-own perform better on product-line variants than on clone-and-own variants.

As product-line variants are inherently consistent, model matchers do not have to rely on complex comparisons because elements that should be matched are often equal. We suspect that this effect can also be observed for other techniques supporting clone-and-own development. As shown in Figure 1 and 2, all matchers appear to perform slightly better on the product-lines *PPU* and *bCMS* than on *Apo-Games*, if no or only few refactorings have been applied.

Hypothesis 2

> Sophisticated techniques supporting clone-and-own yield better results than simple ones.

Advanced methods are mostly developed to break the limitations of ad-hoc or simple solutions. For any amount of refactorings and for each of our experimental subjects, the weight of matchings calculated by *NwM* is higher than the weights obtained by more simple matchers.

Figure 1: Matching Quality With All Refactorings (Setup-A)

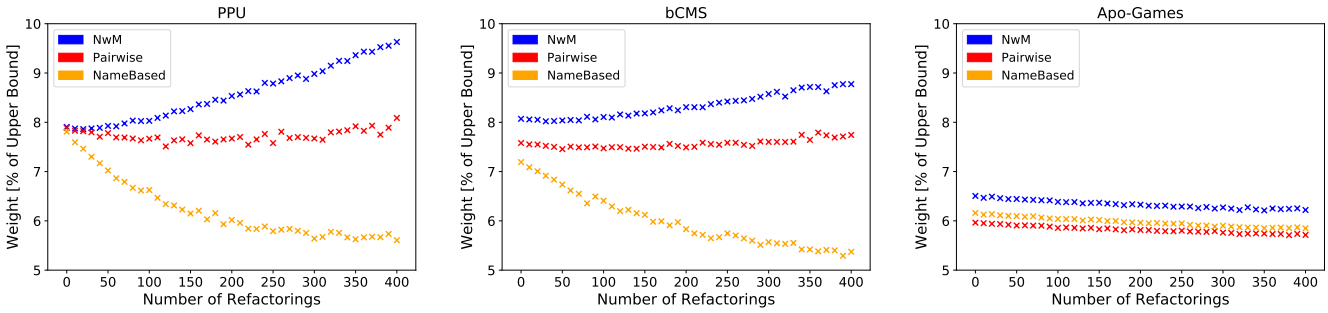
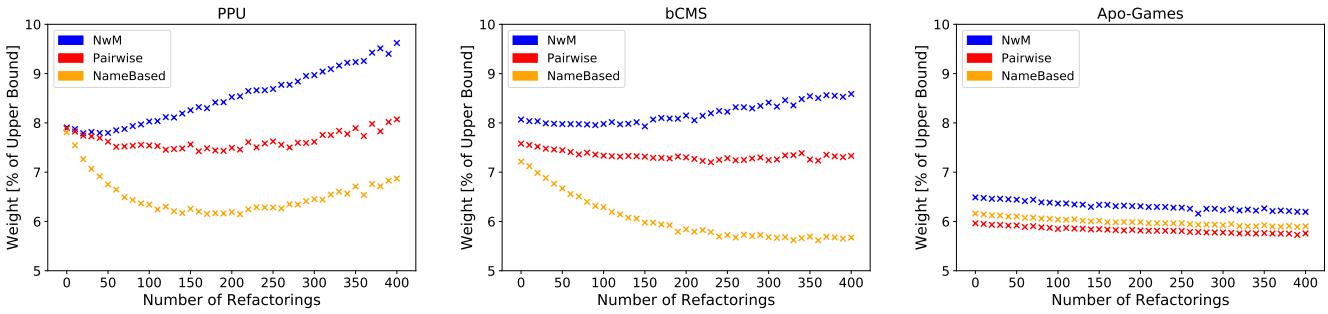


Figure 2: Matching Quality With Only Structural Refactorings (Setup-B)



Hypothesis 3

> Increasing variant drift in product-line variants reveals differences in the quality of results delivered by techniques supporting clone-and-own.

With this hypothesis, we argue that the phenomenon of variant drift can have a strong impact on the behavior of the techniques. While matchers perform equally well for product-line variants (0 refactorings), their performances diverge when simulating variant drift. This effect distinguishes product-line variants (*PPU*, *bCMS*) from clones (*Apo-Games*), where increasing variant drift does not reveal more differences.

Hypothesis 4

> Increasing variant drift in clones does not reveal further differences in the quality of results delivered by techniques supporting clone-and-own.

Clones from clone-and-own development presumably already exhibit variant drift and additional refactoring does not change the relative difference in the quality of results any further. More complex algorithms perform better than simple ones, regardless of the number of additional refactorings. In our experiments, refactorings have not lead to any significant effect for *Apo-Games*.

6 SUMMARY AND FUTURE WORK

At the example of n-way model matching, we investigated how variant drift (i.e., the introduction of unintentional divergences in variants) might affect the empirical evaluation of techniques for the systematic support of clone-and-own development. To simulate variant drift, we systematically applied increasing amounts of refactoring operations to variants of three different experimental subjects, comprising two sets of models stemming from software product lines, and one set of models stemming from a clone-and-own system. Based on our results, we proposed four hypotheses on how variant drift might impact the empirical evaluation of other techniques in clone-and-own research.

Clearly, given the early state of our research reported above, further investigations are required. We want to investigate to which extent variant drift occurs in real world clone-and-own projects, and we plan to conduct additional experiments through which we can either confirm, refine, or even reject our four hypotheses. For instance, we want to extend our evaluation to more experimental subjects, and assess how other techniques applied in clone-and-own development are affected by variant drift. Nonetheless, considering our current results, we call for more realistic experimental subjects that can be used as benchmarks for improving the evaluation of clone-and-own research and reducing bias in external validity.

ACKNOWLEDGMENTS

This work has been supported by the German Research Foundation within the project *VariantSync* (TH 2387/1-1 and KE 2267/1-1).

REFERENCES

[1] Michał Antkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Thomas Schmorleiz, Ralf Lämmel, Ștefan Stănciulescu, Andrzej Wąsowski, and Ina Schaefer. 2014. Flexible product line engineering with a virtual platform. In *International Conference on Software Engineering*. 532–535.

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*.

[3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer.

[4] Johannes Bürdek, Timo Kehrer, Malte Lochau, Dennis Reuling, Udo Kelter, and Andy Schürr. 2016. Reasoning about product-line evolution using complex feature model differences. *Autom. Softw. Eng.* 23, 4 (2016), 687–733. https://doi.org/10.1007/s10515-015-0185-3

[5] Afredo Capozucca, Betty Cheng, Geri Georg, Nicolas Guelfi, Paul Istoan, Gunter Mussbacher, Adam Jensen, Jean-Marc Jézéquel, Jörg Kienzle, Jacques Klein, et al. 2011. Requirements definition document for a software product line of car crash management systems. *ReMoDD repository, at http://www. cs. colostate. edu/remodd/v1/content/bcms-requirements-definition* (2011).

[6] Alfredo Capozucca, Betty Cheng, Nicolas Guelfi, and Paul Istoan. 2011. OO-SPL modelling of the focused case study. In *Comparing Modeling Approaches (CMA) International Workshop affiliated with ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems (CMA@ MODELS2011)*.

[7] Guilherme Cavalcanti, Paulo Borba, Georg Seibt, and Sven Apel. 2019. The Impact of Structure on Software Merging: Semistructured Versus Structured Merge. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1002–1013.

[8] Krzysztof Czarnecki and Ulrich Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*.

[9] Krzysztof Czarnecki and Ulrich W. Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley.

[10] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. 25–34. https://doi.org/10.1109/CSMR.2013.13

[11] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An exploratory study of cloning in industrial software product lines. In *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, 25–34.

[12] Slawomir Duszynski. 2015. *Analyzing similarity of cloned software variants using hierarchical set models*. Fraunhofer IRB Verlag.

[13] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

[14] Christian Kästner, Alexander Dreiling, and Klaus Ostermann. 2013. Variability mining: Consistent semi-automatic detection of product-line features. *IEEE Transactions on Software Engineering* 40, 1 (2013), 67–82.

[15] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. 2014. Propagation of software model changes in the context of industrial plant automation. *at-Automatisierungstechnik* 62, 11 (2014), 803–814.

[16] Benjamin Klatt and Martin Küster. 2013. Improving product copy consolidation by architecture-aware difference analysis. In *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures*. ACM, 117–122.

[17] Benjamin Klatt, Martin Küster, and Klaus Krogmann. 2013. A graph-based analysis concept to derive a variation point design from product copies. In *International Workshop on Reverse Variability Engineering*. 1–8.

[18] Jacob Krüger, Wolfram Fenske, Thomas Thüm, Dirk Aporius, Gunter Saake, and Thomas Leich. 2018. Apo-games: a case study for reverse engineering variability from cloned Java variants. In *Proceedings of the 22nd International Systems and Software Product Line Conference-Volume 1*. 251–256.

[19] Harold W Kuhn. 1955. The Hungarian method for the assignment problem. *Naval research logistics quarterly* 2, 1-2 (1955), 83–97.

[20] Raúl Lapeña, Manuel Ballarin, and Carlos Cetina. 2016. Towards clone-and-own support: locating relevant methods in legacy products. In *International Systems and Software Product Line Conference*. 194–203.

[21] Daniela Lettner and Paul Grünbacher. 2015. Using Feature Feeds to Improve Developer Awareness in Software Ecosystem Evolution. 11:11–11:18. https://doi.org/10.1145/2701319.2701331

[22] Jörg Liebig, Andreas Janker, Florian Garbe, Sven Apel, and Christian Lengauer. 2015. Morpheus: Variability-Aware Refactoring in the Wild. 380–391.

[23] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. 2017. A classification of variation control systems. *ACM SIGPLAN Notices* 52, 12 (2017), 49–62.

[24] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2017. Variability extraction and modeling for product variants. *Software & Systems Modeling* 16, 4 (2017), 1179–1199.

[25] 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 *International Software Product Line Conference*. 101–110.

[26] William F. Opdyke. 1992. *Refactoring Object-Oriented Frameworks*. Ph.D. Dissertation. USA.

[27] Christopher Pietsch, Udo Kelter, Timo Kehrer, and Christoph Seidl. 2019. Formal foundations for analyzing and refactoring delta-oriented model-based software product lines. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A*. 207–217.

[28] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. 2005. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media.

[29] Dennis Reuling, Malte Lochau, and Udo Kelter. 2019. From Imprecise N-Way Model Matching to Precise N-Way Model Merging. *Journal of Object Technology* 18, 2 (2019).

[30] Dennis Reuling, Christopher Pietsch, Udo Kelter, and Timo Kehrer. 2020. Towards projectional editing for model-based SPLs. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*. 1–10.

[31] Julia Rubin and Marsha Chechik. 2013. N-way model merging. In *proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 301–311.

[32] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. 2013. Managing cloned variants: a framework and experience. In *Proceedings of the 17th International Software Product Line Conference*. ACM, 101–110.

[33] Uwe Ryssel, Joern Ploennigs, and Klaus Kabitzsch. 2012. Automatic library migration for the generation of hardware-in-the-loop models. *Science of Computer Programming* 77, 2 (2012), 83–95.

[34] Alexander Schlie, Sandro Schulze, and Ina Schaefer. 2020. Recovering variability information from source code of clone-and-own software systems. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*. 1–9.

[35] Thomas Schmorleiz. 2015. *An Annotation-Centric Approach to Similarity Management*. Master's thesis. Universität Koblenz-Landau, Germany.

[36] Thomas Schmorleiz and Ralf Lämmel. 2014. Similarity Management via History Annotation. In *Seminar Advanced Techniques and Tools for Software Evolution*. Dipartimento di Informatica Università degli Studi dell'Aquila, L'Aquila, Italy, 45–48.

[37] Sandro Schulze, Thomas Thüm, Martin Kuhlemann, and Gunter Saake. 2012. Variant-Preserving Refactoring in Feature-Oriented Software Product Lines. 73–81.

[38] Felix Schwäger and Bernhard Westfechtel. 2016. SuperMod: tool support for collaborative filtered model-driven software product line engineering. In *International Conference on Automated Software Engineering*. IEEE, 822–827.

[39] Ștefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. 2015. Forked and integrated variants in an open-source firmware project. In *International Conference on Software Maintenance and Evolution*. IEEE, 151–160.

[40] Stefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. 2015. Forked and Integrated Variants in an Open-Source Firmware Project. 151–160. https://doi.org/10.1109/ICSM.2015.7332461

[41] Gabriele Taentzer, Timo Kehrer, Christopher Pietsch, and Udo Kelter. 2018. A Formal Framework for Incremental Model Slicing. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, Cham, 3–20.

[42] Carmine Vassallo, Giovanni Grano, Fabio Palomba, Harald C. Gall, and Alberto Bacchelli. 2019. A large-scale empirical exploration on refactoring activities in open source software projects. *Science of Computer Programming* 180 (2019), 1 – 15. https://doi.org/10.1016/j.scico.2019.05.002

[43] Birgit Vogel-Heuser, Alexander Fay, Ina Schaefer, and Matthias Tichy. 2015. Evolution of software in automated production systems: Challenges and research directions. *Journal of Systems and Software* 110 (2015), 54–84. https://doi.org/10.1016/j.jss.2015.08.026

[44] Birgit Vogel-Heuser, Christoph Legat, Jens Folmer, and Stefan Feldmann. 2014. *Researching evolution in industrial plant automation: Scenarios and documentation of the pick and place unit*. Technical Report. Institute of Automation and Information Systems, Technische Universität München.

[45] David Wille, Sandro Schulze, and Ina Schaefer. 2016. Variability mining of state charts. In *Proceedings of the 7th International Workshop on Feature-Oriented Software Development*. ACM, 63–73.

[46] David Wille, Sandro Schulze, Christoph Seidl, and Ina Schaefer. 2016. Custom-tailored variability mining for block-based languages. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 271–282.

[47] 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 *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, 1064–1071.