Apo-Games - A Case Study for Reverse Engineering Variability from Cloned Java Variants

Jacob Krüger
Harz University of Applied Science
Otto-von-Guericke-University
Wernigerode & Magdeburg, Germany
jkrueger@ovgu.de

Wolfram Fenske
Otto-von-Guericke-University
Magdeburg, Germany
wfenske@ovgu.de

Thomas Thüm
Technische Universität Braunschweig
Braunschweig, Germany
t.thuem@tu-braunschweig.de

Dirk Aporius
Eudemonia Solutions AG
Magdeburg, Germany
aporius@eudemonia-solutions.de

Gunter Saake
Otto-von-Guericke-University
Magdeburg, Germany
saake@ovgu.de

Thomas Leich
Harz University of Applied Sciences
METOP GmbH
Wernigerode & Magdeburg, Germany
tleich@hs-harz.de

ABSTRACT

Software-product-line engineering is an approach to systematically manage reusable software features and has been widely adopted in practice. Still, in most cases, organizations start with a single product that they clone and modify when new customer requirements arise (a.k.a. *clone-and-own*). With an increasing number of variants, maintenance can become challenging and organizations may consider migrating towards a software product line, which is referred to as extractive approach. While this is the most common approach in practice, techniques to extract variability from cloned variants still fall short in several regards. In particular, this accounts for the low accuracy of automated analyses and refactoring, our limited understanding of the costs involved, and the high manual effort. A main reason for these limitations is the lack of realistic case studies. To tackle this problem, we provide a set of cloned variants. In this paper, we characterize these variants and challenge the research community to apply techniques for reverse engineering feature models, feature location, code smell analysis, architecture recovery, and the migration towards a software product line. By evaluating solutions with the developer of these variants, we aim to contribute to a large body of knowledge on this real-world case study.

CCS CONCEPTS

• **Software and its engineering** → **Software product lines**; **Software reverse engineering**; *Maintaining software*;

KEYWORDS

Software-product-line engineering, reverse engineering, extractive approach, feature location, case study, data set

1 INTRODUCTION

Software-product-line engineering is a systematic approach to reuse software based on *features* [1, 26]. A feature comprises a user-visible behavior – represented by, for example, models, requirements, and implementation – of the desired software variants. Features are also used as configuration options to instantiate a concrete variant. Consequently, features implement *variable* (shared among some variants) and *common* (shared among all variants) functionalities [1, 19]. To manage a software product line, features and their dependencies are typically modeled with feature models [4, 8, 29] to specify the valid configurations of a software product line.

Despite promising several benefits, such as, reduced development and maintenance costs or faster time-to-market [1, 13, 26], organizations seldom initiate their software as a software product line—fearing the initial investments, uncertainties on the products' future, and corresponding risks [14, 18]. Instead, they often start with a single product that they clone and adopt for new customer requirements, which is referred to as clone-and-own approach [10, 30]. However, with an increasing number of clones, managing and maintaining them can become costly, as most updates and bug-fixes must be propagated to other clones [1, 25, 28]. As a result, organizations later on may consider to migrate towards software-product-line engineering to address these issues, applying the extractive approach [16]. Still, extracting a software product line can be a costly and risky process [7, 18].

These costs and risks often arise because features are not explicitly marked in the source code of the cloned systems and their dependencies as well as interactions are rarely documented. Over time, the developers' memory fades [21] and the variants evolve. Thus, the knowledge about features and their locations diminishes and must be recovered [12, 19]. As a result, *feature location* [3, 9, 27] is one of the most common and most expensive tasks in software engineering [5, 32]—especially as automated techniques often lack in accuracy and must be adapted to the system under investigation [12, 19, 27]. Consequently, for migrating variants towards a software product line, additional costs arise, for example, due to the necessary reverse engineering tasks, the changing development process, and the introduction of variability management [6, 7, 17, 18, 20, 23].

The main problem of automated techniques is their limited applicability in real-world scenarios [5, 12, 19, 20], often resulting from a lack of appropriate case studies to evaluate such techniques against. As a result, these techniques can hardly be tested and adopted for industrial settings. For this purpose, some authors, such as, Olszak and Jorgensen [24], Ji et al. [12], or Krüger et al. [19], provide specific artifact sets containing feature locations. Also, Martinez et al. [22] collect case studies used for different techniques of software-product-line extraction and reverse variability engineering. Despite such efforts, there are few real-world case studies that are publicly

available and provide comparable results for future research. With our contributions, we aim to tackle this problem:

- We contribute a subset of the Apo-Games, making the source code and resources of 20 Java variants – comprising 163.1 KSLOC – and 5 Android variants – comprising 20.9 KSLOC – publicly available on BitBucket.[1] These open-source games are developed by Dirk Aporius based on the clone-and-own-approach. They have evolved over 12 years, are successful in the Android store, are used for student programming competitions at the University of Magdeburg, and have been implemented by an experienced, industrial developer. Thus, we argue that they provide a valid case study for software product line extraction from variants.
- We ask the research community to apply their techniques for reverse engineering on the provided variants. In particular, our challenges are reverse engineering of feature models, feature location, code smell analysis, architecture recovery, and migration to a software product line. Each submitted solution will be evaluated together with the original developer. Thus, in contrast to other studies, we have a single, reliable source of knowledge for the ground truths of the Apo-Games. This is a good opportunity to submit solutions and receive feedback by the real developer.

Overall, we provide a set of real-world variants for reverse engineering variability and extracting software product lines. Submitted solutions will receive feedback and those that reflect the real situation best can serve as ground truths for future research. Thus, we ask to provide open access to any tool and the resulting artifacts in addition to the solution to ensure replicability.

2 APO-GAMES

The Apo-Games are a set of small games that have been implemented by a single developer based on the clone-and-own approach. Since the initial start in 2006 until the end of 2017, three sets of games have evolved:

(1) 40 Java games;
(2) 38 additional Java games comprising less than 4 Kilobyte in byte code and resources; and
(3) 11 Android games.

Overall, the Apo-Games have been quite successful: The Android variants have between 100 and 50.000 downloads each. Several Java games are used at the University of Magdeburg to teach programming in a competitive context. For example, students have to implement search algorithms or small bots that are evaluated against each other.

Over time, several games from different genres have evolved and required adaptations. Some of the underlying concepts in the Apo-Games are inspired by existing games to make them more accessible to students. For example, in Figure 1 we show a screenshot of the ApoMario game, which is a platform game inspired by the *Super Mario* series. Other game types include a football simulation (i.e., ApoSoccer) and riddles (e.g., ApoImp).

[1]Source code: https://bitbucket.org/Jacob_Krueger/apogamessrc
Games in action (in German): http://apo-games.de/.

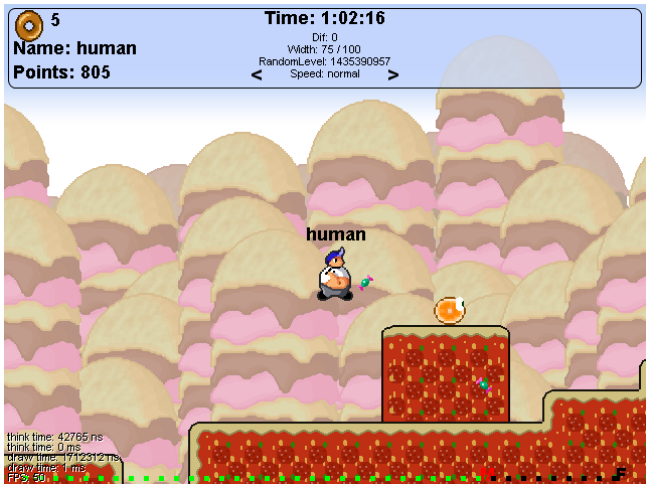

Figure 1: Screenshot of ApoMario.

Case Study Variants With this challenge, we contribute a larger dataset of 20 Java and 5 Android games that can be used for reverse engineering approaches on cloned variants. Considering the Java variants, all of these games have been developed before 2013, when an evolutionary framework change resulted in major adaptations in the implementation. Thus, games developed afterwards can hardly been seen as clones of such earlier Apo-Games anymore. While some of the Apo-Games are written with German user interfaces and German comments, all identifiers in the source code itself are in English.

We summarize the corresponding set, name, development year, and source lines of code of all 25 games in Table 1. Here, we can see that the games have been developed over 8 years and evolved from each other. The contributed games comprise between 1.7 KSLOC and 19.6 KSLOC. Individually, they are relatively small but cumulate to overall sizes of 163.1 and 20.9 KSLOC. As they are developed by a single developer, the games have some specific characteristics that can help or pose problems during the challenges. For example, there are specific naming conventions, meaning that cloned classes may contain the game title as a prefix in their name.

3 THE CHALLENGES

The Apo-Games are subject systems for reverse engineering variability from cloned variants. In this section, we describe five challenges that we consider interesting and that will help to provide necessary reference artifacts for further research. We focus on *reverse engineering feature models*, *feature location*, *code-smell and impact analysis*, *architecture recovery*, as well as *migration to a software product line*. While each challenge can be addressed separately, they are connected and we encourage researchers to select a suitable subset of their choice.

For each challenge, we are interested in manually, semi-automatically, or fully-automatically derived solutions. It is important that all steps, problems, and results are well documented to ensure that they can be evaluated and compared to each other. This should also include reporting and discussing experiences and lessons learned

Table 1: Details of the contributed Apo-Games.

Set	Name	Year	SLOC
Java	ApoCheating	2006	3,960
	TutorVolley	2006	1,659
	ApoDefence	2007	12,917
	ApoSkunkman	2007	8,645
	ApoStarz	2008	6,454
	ApoBot	2009	5,857
	ApoSoccer	2009	10,736
	ApoCommando	2010	9,820
	ApoIcejumpReloaded	2010	8,138
	ApoPongBeat	2010	6,591
	ApoIcarus	2011	5,851
	ApoMarc	2011	5,493
	ApoMario	2011	17,184
	ApoSlitherLink	2011	7,313
	ApoNotSoSimple	2011	7,558
	ApoRelax	2011	6,868
	ApoSimple	2011	19,558
	ApoSnake	2012	6,557
	ApoSudoku	2012	5,517
	ApoImp	2012	6,432
	Sum	20	163,108
Android	ApoClock	2012	3,615
	ApoDice	2012	2,523
	ApoSnake	2012	2,965
	ApoMono	2013	6,487
	myTreasure	2013	5,360
	Sum	5	20,950

for each solution. For all tools that are used, we ask that they are publicly available. However, these do not need to be tools specifically developed for software-product-line engineering. In addition, we ask that any created artifact, for example, feature models or source code with feature locations, is made publicly available. Both, tools and artifacts, have to be available from the point of submitting a solution henceforth. We also remark that it is not necessary to include all variants into a solution. However, it has to be justified why specific variants are not considered.

3.1 Reverse Engineering a Feature Model

A common problem of software-product-line adoption is to identify features and their constraints in legacy systems, and to derive a feature model from these. This often requires specific input artifacts, domain knowledge, and manual analysis. For example, developers may analyze existing documentation to identify feature descriptions based on which they reverse engineer constraints and a model. Some approaches assume either a feature for each code clone that is shared among variants or a feature for each variant. However, these are hardly actual features – neither in a notion of functionality nor in a notion of variability.

Task We ask for solutions that describe how a feature model can be derived from the source code of cloned variants. Consequently, the applied process and its single steps should be reported. Finally,

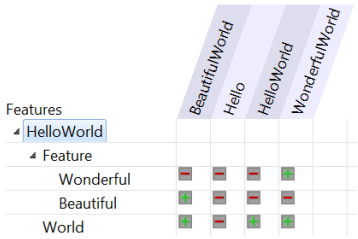

Figure 2: Configuration map for a Hello-World example from FeatureIDE.

a feature model for the Apo-Games variants shall exist that defines the features and their dependencies. Additionally, it should be explained how meaningful the identified features are.

Evaluation For evaluation purposes, we expect, in any format that can be imported by FeatureIDE [15, 31], the feature model (e.g., in FeatureIDE, GUIDSL, SXFM, Velvet, or DIMACS format) of the Apo-Games, and further statistics. These statistics include the numbers and types of features as well as their distribution among the original variants. The distributions should be provided in a comprehensive overview that can be, or should resemble, a configuration map in FeatureIDE, for which we show an example in Figure 2. In addition, the degree of granularity that is used in the variability model will be evaluated: One feature for each variant does not help, as it is likely to be too coarse-grained and one feature for every differing statement is probably too fine grained. Here, we expect a proper explanation how a certain granularity is defined and ensured within the proposed solution.

3.2 Feature Location

Feature location [27] is a common problem in maintaining and reverse engineering software, often connected to high efforts even for a single system. The particular challenge is to locate only the source code that belongs to specific features. To this end, at least some features need to be identified and the results must be documented. In the context of cloned systems, these challenges increase as each feature can be present in multiple variants with certain variations.

Task Solutions to this challenge shall provide the source code of several variants in which features are mapped. We encourage to support multiple representations, for example, by annotating features in the source code. Still, we require one artifact that lists each feature and the lines of code that belong to it in each variant. It should be discussed how these feature locations are found and separated from each other. In particular, variations of features in different variants and their characteristics, such as size, tangling, and scattering, are interesting to compare among variants.

Evaluation Considering the evaluation, we ask that the locations of each feature are mapped and saved persistently within separate csv-files. It has to be explained which features have been located based on which approach. At least, the following metrics have to be presented to evaluate a solution:

- *Size and distribution* represent the source lines of code (SLOC) of a feature. For this purpose, the locations of a feature should be documented with an identifier, to relate them among variants, as well as start and end line in each variant and file.

- *Scattering degree* measures at how many locations consecutive lines of code in a variant implement a feature.
- *Tangling degree* describes the number of other features that are part of the considered feature.

For the distribution, we recommend a csv-file for each feature. Furthermore, another csv-file should show the scattering, tangling, and sizes of features among variants to show variations and distributions. In addition, we recommend to discuss the quality of the results and the necessary efforts to obtain them.

3.3 Code Smell Analysis

Code smells indicate flaws in the design and implementation of software [11]. Such flaws can be harmful as they potentially lead to bugs or challenge program comprehension. Consequently, they can already be problematic in a single system. However, in cloned variants they may be even more problematic, as they appear in several systems – meaning that removing them requires change propagation. Also, addressing code smells may be a reason to adopt software-product-line engineering, but they can also complicate the adoption process.

Task Any solution to this challenge has to identify code smells in the Apo-Games and must describe the applied process. This includes not only analysis of a single system but also identifying the located smells in other variants. In addition, we ask that solutions assess the impact of the identified code smells on maintenance or software-product-line adoption. To this end, efforts to resolve identified smells – within the clones or during migrations – have to be estimated and discussed.

Evaluation Solutions for this challenge have to include several statistics that should also be provided as a separate csv-file for each variant, including the following columns:

- *Type* describes the type of code smell that has been identified.
- *Identifiers* should be used to identify the exact same smell within the same and among multiple variants.
- *File* refers to a file that is affected by a code smell.
- *Positions* should include the affected lines of code, for example, by providing line numbers or start and end line, that contribute to the code smell.

As for feature location, a summarizing csv-file should show the total number and distribution of code smells among the variants. Optionally, we are also interested in understanding why a smell may not exist in other variants, even if the same or changed code is present.

3.4 Architecture Recovery

Architectural views offer a coarse-grained perspective on a software system. Such views help to understand a system's inner workings by hiding distracting details. As most likely accounts for many cloned systems, manual inspection of the Apo-Games indicates that many of them follow a similar structure and, thus, may have similar architectures. Despite many similarities, there are also some derivations not only in terms of features (a functional perspective) but also code files, their structure, and their dependencies (an architectural perspective). Already having an architecture view on a software product line in terms of components and classes can be a powerful guide for their adoption. The challenge is to create this architecture based on an analysis of the variants.

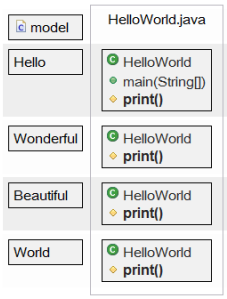

Figure 3: Collaboration diagram for a Hello-World example from FeatureIDE.

Task We ask the participants of this challenge to reconstruct and consolidate the architecture of the Apo-Games into aggregated UML class diagrams, potentially including variability in attributes and methods. The goal is to create a unified architectural view of the selected variants, which could be used to migrate them into a software product line. Along with aggregated architectures, participants are requested to submit a description of their process and tool chain. Moreover, efforts as well as challenges – both, addressed and open – should be documented. Optionally, an estimation of the effort to condense single classes can be provided.

Evaluation We require aggregated UML class diagrams, including important classes and their identified dependencies. In the context of software-product-line engineering, important refers especially to variability and core classes, which can be clustered if necessary. Each modeled class has to be mapped to the variants it contributes to, for example, using a representation in the UML diagram, a separate collaboration diagram [1] (cf. Figure 3), or a similar representation. As concrete metrics, we ask for each class to provide the number of variants it contributes to, as well as the differences in code size, number of attributes, and number of methods. For the estimated migration effort, we expect that potential costs are exemplified based on the available data. Still, we do not ask to use a cost model but to reason about relative costs of migrating specific classes – considering identical, adapted, or unique parts among variants.

3.5 Migration to a Software Product Line

Code duplication is the number one problem arising from clone-and-own development, requiring updates in all variants. One of the aims of migrating variants to a software product line is therefore to reduce the amount of duplicated code (i.e., code clones). Specifically, the goal is not to reduce code clones within a single product, but to reduce code clones between multiple variants. To this end, features are extracted from the variants.

Task In this challenge, we ask for a migration of the provided variants into a software product line. Considering the extraction process, the focus should be to describe a detailed process including the applied tasks, identified problems, and necessary efforts, for example, by measuring the required time for and the number of performed activities. In the end, a software product line shall exist

that includes mandatory and optional features from the variants and can be used to instantiate these. For the variability mechanism, any implementation technique can be used. However, we recommend those supported by FeatureIDE to facilitate the evaluation, for example, Antenna, FeatureHouse [2], frameworks, or runtime parameters.

Evaluation To evaluate a solution, we require the extracted software product line. As particular results, we expect measurements on the amount of code clones that have been removed from the variants. This also includes numbers and sizes of derived features in the software product line. Another evaluation is to show to which extend the cloned variants and their extracted software product line counterparts are identical. Thus, each solution has to show that the variants generated in the software product line suit the original ones. Considering the experiences and lessons learned, we ask that they are consolidated and compared to existing works.

4 SUMMARY

In this paper, we described the Apo-Games, a set of cloned Java and Android games. Of these, we contribute the sources of 20 Java and 5 Android variants in a repository and challenge the research community to propose solutions that describe these variants. To condense a ground truth for further research, we call for feature models, feature location, knowledge on code-smells, architectural views, and extracted software product lines. All solutions will be evaluated with the original developer and we will include proper solutions into the repository to provide artifacts to evaluate and compare future approaches against.

ACKNOWLEDGMENTS

This research is supported by DFG grants LE 3382/2-1, SA 465/49-1, and Volkswagen Financial Services AG.

REFERENCES

[1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.
[2] Sven Apel, Christian Kastner, and Christian Lengauer. 2009. FeatureHouse: Language-Independent, Automated Software Composition. In *International Conference on Software Engineering*. IEEE, 221–231.
[3] Wesley Klewerton Guez Assunção and Silvia Regina Vergilio. 2014. Feature Location for Software Product Line Migration: A Mapping Study. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 52–59.
[4] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Transactions on Software Engineering* 39, 12 (2013), 1611–1640.
[5] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. 1986. The Concept Assignment Problem in Program Understanding. In *International Conference on Software Engineering (ICSE)*. IEEE, 482–498.
[6] Günter Böckle, Jesús Muñoz, Peter Knauber, Charles Krueger, Julio do Prado Leite, Frank van der Linden, Linda Northrop, Michael Stark, and David Weiss. 2002. Adopting and Institutionalizing a Product Line Culture. In *International Conference on Software Product Lines (SPLC)*. Springer, 1–8.
[7] Paul C. Clements and Charles W. Krueger. 2002. Point/Counterpoint: Being Proactive Pays Off/Eliminating the Adoption Barrier. *IEEE Software* 19, 4 (2002), 28–30.
[8] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. 2012. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 173–182.
[9] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature Location in Source Code: A Taxonomy and Survey. *Journal of Software: Evolution and Process* 25, 1 (2013), 53–95.
[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. In *European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 25–34.
[11] Martin Fowler and Kent Beck. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.
[12] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. 2015. Maintaining Feature Traceability with Embedded Annotations. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 61–70.
[13] Peter Knauber, Jesus Bermejo, Günter Böckle, Julio Cesar Sampaio Do Prado Leite, Frank van der Linden, Linda M. Northrop, Michael Stark, and David M. Weiss. 2002. Quantifying Product Line Benefits. In *International Workshop on Product-Family Engineering (PFE)*. Springer, 155–163.
[14] Peter Knauber, Dirk Muthig, Klaus Schmid, and Tanya Widen. 2000. Applying Product Line Concepts in Small and Medium-Sized Companies. *IEEE Software* 17, 5 (2000), 88–95.
[15] Sebastian Krieter, Marcus Pinnecke, Jacob Krüger, Joshua Sprey, Christopher Sontag, Thomas Thüm, Thomas Leich, and Gunter Saake. 2017. FeatureIDE: Empowering Third-Party Developers. In *International Systems and Software Product Line Conference*. ACM, 42–45.
[16] Charles W. Krueger. 2002. Easing the Transition to Software Mass Customization. In *International Workshop on Software Product-Family Engineering (PFE)*. Springer, 282–293.
[17] Jacob Krüger. 2017. Lost in Source Code: Physically Separating Features in Legacy Systems. In *International Conference on Software Engineering (ICSE)*. IEEE, 461–462.
[18] Jacob Krüger, Wolfram Fenske, Jens Meinicke, Thomas Leich, and Gunter Saake. 2016. Extracting Software Product Lines: A Cost Estimation Perspective. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 354–361.
[19] Jacob Krüger, Wanzi Gu, Hui Shen, Mukelabai Mukelabai, Regina Hebig, and Thorsten Berger. 2018. Towards a Better Understanding of Software Features and Their Characteristics: A Case Study of Marlin. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 105–112.
[20] Jacob Krüger, Louis Nell, Wolfram Fenske, Gunter Saake, and Thomas Leich. 2017. Finding Lost Features in Cloned Systems. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 65–72.
[21] Jacob Krüger, Jens Wiemann, Wolfram Fenske, Gunter Saake, and Thomas Leich. 2018. Do You Remember This Source Code?. In *International Conference on Software Engineering (ICSE)*. ACM. To Appear.
[22] Jabier Martinez, Wesley K. G. Assunção, and Tewfik Ziadi. 2017. ESPLA: A Catalog of Extractive SPL Adoption Case Studies. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 38–41.
[23] Linda M. Northrop. 2002. SEI's Software Product Line Tenets. *IEEE Software* 19, 4 (2002), 32–40.
[24] Andrzej Olszak and Bo Norregaard Jorgensen. 2011. Understanding Legacy Features with Featureous. In *Working Conference on Reverse Engineering (WCRE)*. IEEE, 435–436.
[25] Tristan Pfofe, Thomas Thüm, Sandro Schulze, Wolfram Fenske, and Ina Schaefer. 2016. Synchronizing Software Variants with VariantSync. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 329–332.
[26] Klaus Pohl, Günter Böckle, and Frank van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer.
[27] Julia Rubin and Marsha Chechik. 2013. A Survey of Feature Location Techniques. In *Domain Engineering*. Springer, 29–58.
[28] Julia Rubin, Andrei Kirshin, Goetz Botterweck, and Marsha Chechik. 2012. Managing Forked Product Variants. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 156–160.
[29] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. 2012. Software Diversity: State of the Art and Perspectives. *International Journal on Software Tools for Technology Transfer* 14, 5 (2012), 477–495.
[30] Ştefan 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 (ICSME)*. IEEE, 151–160.
[31] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming* 79 (2014), 70–85.
[32] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. 2013. How Developers Perform Feature Location Tasks: A Human-Centric and Process-Oriented Exploratory Study. *Journal of Software: Evolution and Process* 25, 11 (2013), 1193–1224.