Simulating the Evolution of Clone-and-Own Projects with VEVOS

Alexander Schultheiß
Humboldt University of Berlin
Germany
schultha@informatik.hu-berlin.de

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

Sascha El-Sharkawy
University of Hildesheim
Germany
elscha@sse.uni-hildesheim.de

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

Timo Kehrer
University of Bern
Switzerland
timo.kehrer@inf.unibe.ch

ABSTRACT

In clone-and-own development, new variants of a software system are typically created by manually copying and adapting an existing variant. This approach is flexible but suffers from various challenges such as high maintenance cost in the long term. While researchers started to address the challenges of clone-and-own, there is yet little empirical evidence on the efficiency and effectiveness of clone-and-own research. The main reason for this is the lack of appropriate benchmarks, which need to expose a multitude of different data and meta-data serving as input and ground truth for experimental evaluations. We present VEVOS, a benchmark generation framework that picks up these requirements and, given the version history of a software product line, enables the simulation of the evolution of cloned variants, and provides meta-data serving as ground truth.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**.

KEYWORDS

Clone-and-own, software product lines, experimental subjects, benchmarks, empirical evaluation

ACM Reference Format:
Alexander Schultheiß, Paul Maximilian Bittner, Sascha El-Sharkawy, Thomas Thüm, and Timo Kehrer. 2022. Simulating the Evolution of Clone-and-Own Projects with VEVOS. In *The International Conference on Evaluation and Assessment in Software Engineering 2022 (EASE 2022), June 13–15, 2022, Gothenburg, Sweden.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3530019.3534084

1 INTRODUCTION

Variability is a key requirement for today's software to meet varying customer expectations or enable mass-customization. The state-of-practice in engineering multi-variant software systems often follows a simple pattern: The development starts with a single variant and further variants are added later by copying and adapting an existing variant; a principle which is generally known as clone-and-own [3, 12, 45, 51]. While clone-and-own has the short-term benefits of minimal cost and time-to-market, it causes high maintenance costs in the long run [3, 12, 31, 51]. For example, when fixing a bug in one variant, it is unknown which other variants are affected by the same bug and thus need to be fixed accordingly.

Given these practical experiences, a more recent line of research focuses at better supporting clone-and-own [8, 11, 19, 21, 24, 31, 34, 37, 39, 48]. Clone-and-own research focuses on developing methods and tools with the ultimate goal of either migrating a clone-and-own project to a software product line, or systematically supporting clone-and-own development through better automation. For example, the identification of common and individual parts of cloned variants [13, 16] (aka. variablity extraction) serves as a preparatory step for migration, while tracing features to development artifacts [2, 8, 19] (aka. feature trace recording) serves as a foundation for automatically synchronizing cloned variants [24].

To date, however, there is little empirical evidence on the efficiency and effectiveness of such clone-and-own support when applied to real-world projects. The main reason for this is the lack of appropriate benchmarks [50], which must expose a multitude of different data and meta-data. For example, next to the source code of a set of cloned variants serving as input, the empirical evaluation of variability extraction [13, 16] also requires a *code matching* as ground truth, linking corresponding code entities among variants. Likewise, an empirical evaluation of feature trace recording [2, 8, 19] requires the condition under which each code artifact is included in a variant (aka. presence condition) at each revision.

Existing benchmarks and study subjects are hardly applicable to empirically evaluate clone-and-own support [50]. On the one hand, real-world clone-and-own projects lack the required meta-data that can serve as ground truth because cloned variants are created ad-hoc. The manual reverse engineering of these meta-data as proposed by Stănciulescu et al. [51] is tedious, prone to errors, and even infeasible for the entire development history of a larger project. On the other hand, one could use the history of a software product line for generating suitable benchmarks. However, for the two real-world software product lines which have been most widely used for the sake of empirical evaluation in the area of product-line testing and analysis, namely Linux [15, 25, 28, 33, 42] and

EASE 2022, June 13–15, 2022, Gothenburg, Sweden

ACM ISBN 978-1-4503-9613-4/22/06.
https://doi.org/10.1145/3530019.3534084

BusyBox [17, 25, 30, 32, 43], the product-line's feature model and presence conditions are hidden behind the build system and preprocessor macros. Feature model and presence conditions must be automatically reverse engineered, and transferred into variant-specific data and meta-data.

In this paper, we present a benchmark generation framework called VEVOS (Variant EVOlution Simulation) that solves these technical and conceptual challenges (cf. Sec. 5). VEVOS enables to simulate the evolution of clone-and-own projects which can be used as benchmarks for empirical research. To the best of our knowledge, the generated benchmarks are the first ones that provide all the required data and meta-data across the history of an evolving project (cf. Sec. 3). Thereby, we pave the way for collecting empirical evidence on the efficiency and effectiveness of clone-and-own support. We demonstrate the feasibility of our approach by instantiating VEVOS for Linux and BusyBox (cf. Sec. 6).

2 MULTI-VARIANT SOFTWARE SYSTEMS

Software Product Lines. A software product line is a set of similar software variants (aka. products) with well-defined commonalities and variability, developed based on a common code base. Software product-line engineering is distinguished into domain engineering and application engineering [4, 9, 44].

Domain engineering is concerned with the specification and implementation of the conceptual features comprised by a product line. Typically, a feature model [5, 9] defines the set of features and their valid combinations (aka. configurations) by specifying their constraints. The specified features are then implemented in a common code base by choosing a variation mechanism which specifies which code belongs to which features. One widely used mechanism are C-preprocessor macros (e.g., `#if`, `#ifdef`, and `#ifndef`) that are embedded into source files and determine which lines of code are included when compiling the file, thereby dividing the code into blocks with individual *block conditions*. By nesting code blocks, annotations implement interactions between features, meaning that code is included or excluded if specific combinations of block conditions are fulfilled. Next to code annotations, build files may contain *file conditions* that in- or exclude entire source files into or from compilation [4]. Such compositions of block and file conditions are known as *presence conditions* of code [4].

Application engineering refers to the generation of a variant based on a valid configuration. Given such a configuration, the product-line's build system can derive a corresponding variant from the common code base by interpreting the presence conditions.

Clone-and-Own. Software projects in practice often rely on an alternative approach to developing multi-variant systems: clone-and-own development. The clone-and-own workflow is as easy as copying and adapting existing variants. Fig. 1 presents an example of a project with three variants. For the sake of the example, we highlight code implementing specific features, but note that this information is usually not available. An existing variant, called V_0, is cloned by branching and extended by the implementation of the feature *Debug* to form a new variant V_1. Later, V_1 is branched to yield variant V_2. Both V_1 and V_2 undergo further changes such that they eventually implement the features *Network* and *Print*, respectively. In parallel, V_0 has also evolved by editing code that

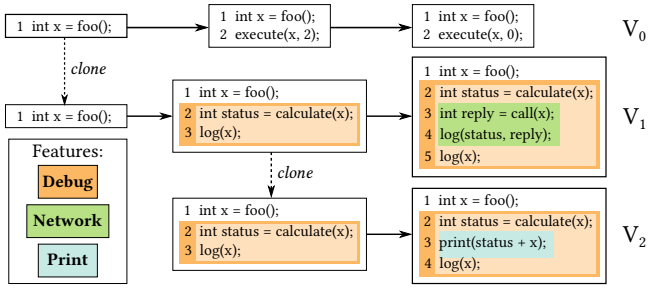

Figure 1: Clone-and-own project with three variants.

does not belong to a particular feature. Thus, the edit still has to be propagated to variants V_1 and V_2.

While, from a short-term perspective, clone-and-own development is more flexible than introducing a software product line, it suffers from maintenance problems in the long run [3, 12, 31, 51]. Clone-and-own research tries to address the challenges of clone-and-own and different approaches have been proposed over the years. The approaches range from the systematic support of clone-and-own development (with the aim of reducing maintenance costs) [8, 11, 19, 24, 31, 34, 48] to techniques that support the migration of a set of variants into an integrated platform (with the aim of eliminating duplicated code) [11, 21, 39]. To evaluate these approaches, suitable benchmarks are required.

3 PROBLEM STATEMENT

Benchmark Requirements. Based on our own experience, we found several requirements for a benchmark to be suitable for the evaluation of clone-and-own research. In any case, we need (i) the source code of a set of variants exposing commonalities and differences due to implementing different combinations of a common set of features. More specifically, we not only need a single snapshot of these variants, but (ii) a version history that represents their parallel evolution. In addition, we often need a multitude of different meta-data serving as additional input or ground truth for empirical evaluation. This includes, for every snapshot in the history of a set of cloned variants, (iii) a feature model that captures the space of valid configurations; (iv) the configuration of each individual variant, (v) presence conditions for the code of each individual variant, and (vi) a matching that identifies the corresponding source code entities among all variants. We summarize (iv) and (v) by the notion that cloned variants need to be *feature-aware*.

The Current State of Benchmarks. Similar requirements and an even more detailed analysis of individual evaluation scenarios are presented by Strüber et al. [50]. In a recent survey on benchmarks for the empirical evaluation of techniques, they found that the overall applicability of existing benchmarks is low, and that certain scenarios that are considered particularly relevant (e.g., variant synchronization) have no applicable benchmark at all.

The Quest for Benchmark Generation. With the goal of generating suitable benchmarks from the history of a software product line, we considered the ESPLA catalog [35], a large collection of case studies from the field of extractive software product-line research. However, we found that only few subject systems fulfill

the prerequisites of benchmark generation. As of January 2022, the catalog comprises a total of 135 case studies of which 45 comprise publicly available subjects. After considering the requirement of the subjects comprising source code with commonalities and differences, only 27 studies remain. Only three of these studies come with a source code history: the Linux kernel, Busybox, and uClibc.

However, the remaining three subjects are preprocessor-based software product lines without an explicit feature model and explicit presence conditions (i.e., presence conditions are only given implicitly through the nesting of annotated code blocks). Thus, extracting a ground truth requires additional tooling.

Ground Truth Extraction. We can employ static product-line analyses to extract a ground truth from a software product line. For example, considering annotation-based product lines, information about the variability such as block conditions, presence conditions, and feature models can be extracted and analyzed. A feature model can be extracted from the build system by analyzing which features are defined and which constraints exist between them.

Yet, a software product line, its presence conditions, and its feature model do not constitute a benchmark. A benchmark also requires the source code of variants to be feature-aware, and a matching between the variants' source code entities.

Variant Generation. The build systems of real-world subject systems (e.g., *Kbuild* used in Linux and BusyBox) are not designed to derive the source code of a variant. Considering C-preprocessor-based product lines, one could consider to make use of the preprocessor to resolve the variability defined by its macros. Executing the C-preprocessor to derive a variant is infeasible, because all macros are resolved – even the ones that do not correspond to variability such as `#include` statements. The results are source files bloated to several times their original size. Such source files hardly correspond to real clone-and-own variants, in which modularization of source code into several files is kept intact.

A solution to this problem could be provided by partial preprocessors such as *cppp*[1] or *coan*,[2] which resolve only specified macros. Nevertheless, the problem of feature-awareness of variants remains: Each variant has a different subset of the product line's code, thus the presence conditions of the product line do not align with a variant's source code, and cannot be used as ground truth for a variant. A benchmark requires variant-specific presence conditions.

4 RELATED WORK

Benchmarks. As discussed in Sec. 3, Strüber et al. [50] found existing benchmarks to have low applicability to clone-and-own research. Most of these benchmarks either lack a version history [23, 29, 36], or comprise only few isolated versions [1, 38, 46, 52]. The DoSC [53] dataset comprises no variants, but only histories of independent subject systems. Berger et al. [7] collected a set of 128 feature models from open-source projects, lacking source code and presence conditions. Lastly, the ClaferWebTools benchmark comprises academic projects developed by students [19].

A more recent benchmark by Michelon et al. [40] targets the evolution of systems in time and space [49] (i.e., version history and existing configurations). There are two classes of variant sets in their benchmark: Sets addressing evolution in space and sets addressing evolution in time. However, the provided ground truth has no feature model and contains only partial presence conditions; they resolve the nesting of block conditions but lack file conditions.

Ground Truth Extraction. GOLEM [10], is an analyzer for variability-related bugs in software. GOLEM probes the build system with build requests for each feature and then determines which files would have been included [10]. However, GOLEM thereby requires more than 90 minutes to process a single version of Linux [10], rendering the extraction of a representative range of versions infeasible (i.e., several thousand commits).

TypeChef [22] is a variability-aware parser developed for static type checking of variable code. However, TypeChef expands preprocessor macros not related to variability, thereby expanding the source files to several times their size by resolving `#include` macros, no longer representing realistic source code.

KernelHaven [27] is an experimentation workbench for static software product line analyses (e.g., the detection of feature effects [41] or the calculation of code metrics [14]). *KernelHaven* follows a modular design relying on plugins that are responsible for extracting and analyzing knowledge about a product line; there are plugins for extracting block conditions, presence conditions of files, and feature models. While *KernelHaven* was built to analyze only isolated versions of a system, its plugin infrastructure was designed to be extendable. We thus utilize the available extraction capabilities by extending *KernelHaven* and integrating it into `VEVOS`.

5 BENCHMARK GENERATION WITH VEVOS

To address the lack of benchmarks, we propose `VEVOS`: a framework for the generation of benchmarks that can be used for the evaluation of techniques supporting multi-variant system development. In this work, we instantiate VEVOS for Busybox and Linux, while its framework infrastructure allows to support further product lines in the future. Because extracting a ground truth takes considerably more time than simulating variants, `VEVOS` is divided into two modules, the ground truth extraction (cf. Sec. 5.1) and the variant simulation (cf. Sec. 5.2).

5.1 Ground Truth Extraction

An overview of `VEVOS`' ground truth extraction is shown in Fig. 2. The extraction of VEVOS internally relies on the *KernelHaven* framework [27, 47] but (1) extends it with a new plugin to analyze presence conditions, and (2) wraps *KernelHaven* within new services to process the entire history of the input product line.

First, the required environment is prepared in a Docker container (0), such that the ground truth extraction can be executed on any system supporting Docker. After preparation, the caller of the extraction specifies a range of revisions that should be considered. For this specified range, the version-control system (VCS) service retrieves all revisions in the range (1); The VCS service is responsible for managing the product line's repository (e.g., checkout revisions), and it cleans the product line's files after presence conditions and a feature model have been extracted. Starting from the first commit in the commit range, the VCS service iteratively conducts a checkout of each revision (1).

[1] https://www.muppetlabs.com/~breadbox/software/cppp.html
[2] http://coan2.sourceforge.net/index.php?page=about

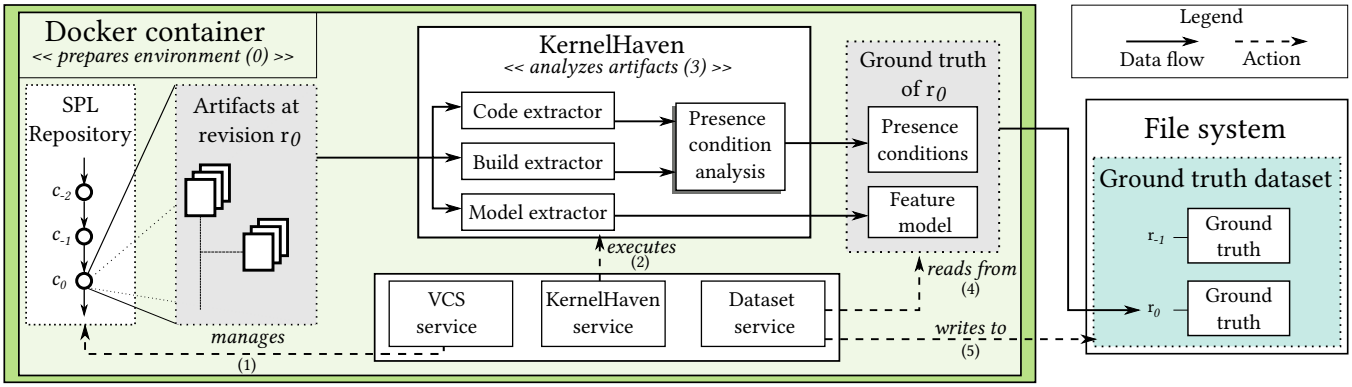

Figure 2: Overview of the ground truth extraction of VEVOS.

For each revision, our *KernelHaven* service is responsible for the configuration and execution of *KernelHaven* which analyzes the artifacts (3) to extract the presence conditions and feature model (i.e., the ground truth) of the current revision. The code model extractor analyzes variability in the source code files yielding code blocks and their respective block conditions (cf. Sec. 2). The build model extractor determines the presence condition of each source file by analyzing the build files of the product line. The feature model extractor collects features and constraints defined by the build system. Based on the extractors' output, our presence condition analysis determines the code's presence conditions by combining block conditions with the file conditions of source files.

Once presence conditions and a feature model have been extracted, our *dataset service* collects (4) and packages (5) them into an archive that is added to the ground truth dataset. The dataset comprises a ground truth for each processed revision. It thereby reflects the evolution of the product line over time, and can be used to simulate the evolution of clone-and-own variants.

5.2 Variant Simulation

The variant simulation is a library for simulating the evolution of clone-and-own variants based on a ground truth dataset and the repository of the corresponding software product line, thereby providing benchmarks for clone-and-own research. As shown in Fig. 3, the library offers three core functionalities: The loading of a ground truth dataset, the sampling of variant configurations from a feature model, and the generation of feature-aware variants.

First, the dataset loading component exposes convenience methods for loading the various types of data and thus provides access to the ground truth when evaluating clone-and-own research.

Second, the configuration sampling component offers functionality for deriving configurations from feature models. The configuration sampling component offers different sampling strategies, for example, random sampling for each revision, or using a predefined set of configurations. Internally, the library calls the *FeatureIDE*-library [26], which contains a variety of functionality for dealing with feature models, e.g., import and export of models, and the generation of valid configurations for a given model.

Third, the variant generation derives feature-aware variants. Each feature-aware variant comprises source code, the variant's configuration (i.e., the list of features that it implements), presence conditions mapped to the code, block and file conditions (aka. feature mappings), and a matching of the variant's code to the product line's code (i.e., line numbers are matched).

In summary, VEVOS fulfills all requirements to benchmarks discussed in Sec. 3: Researchers can use the variant simulation library to generate benchmarks based on a ground truth dataset; these benchmarks comprise the evolution of feature-aware variants for a desired range of revisions; and feature-aware variants comprise the required data and meta-data to evaluate clone-and-own approaches. By providing the sampled configurations and extracted ground truth dataset in their replication package, researchers can comply to open science principle, making their research results replicable and comparable to others.

6 VALIDATION OF TECHNICAL FEASIBILITY

In this section, we validate the technical feasibility of generating suitable clone-and-own benchmarks with VEVOS, We implemented VEVOS in two separate projects (i.e., extraction[3] and simulation[4]).

6.1 Ground Truth Extraction

We validate the extraction's feasibility by executing it for the histories of Linux and BusyBox.[5]

Extraction Setup. We select the following three extractor plugins used in *KernelHaven*: *CodeBlockExtractor* for extracting a code model by retrieving code blocks through parsing preprocessor macros (see Sec. 3), *KbuildMinerExtractor* for extracting a build model by analyzing the build process of files in *Kbuild* with *kbuildminer* [6], and *KconfigReaderExtractor* for extracting a variability model by detecting existing features and constraints between them with *KConfigReader* [20]. *CodeBlockExtractor* was recommended by *KernelHaven*'s developers; while the *KbuildMinerExtractor* and *KconfigReaderExtractor* plugins are currently the only publicly available build and feature model extractors.

Validation Results. For BusyBox, we ran the ground truth extraction for the entire version history and were able to extract a

[3]https://github.com/VariantSync/VEVOS_Extraction
[4]https://github.com/VariantSync/VEVOS_Simulation
[5]System: i7-8700K @3.70Ghz CPU; 32 GB DDR4 RAM; 1TB M.2 SSD

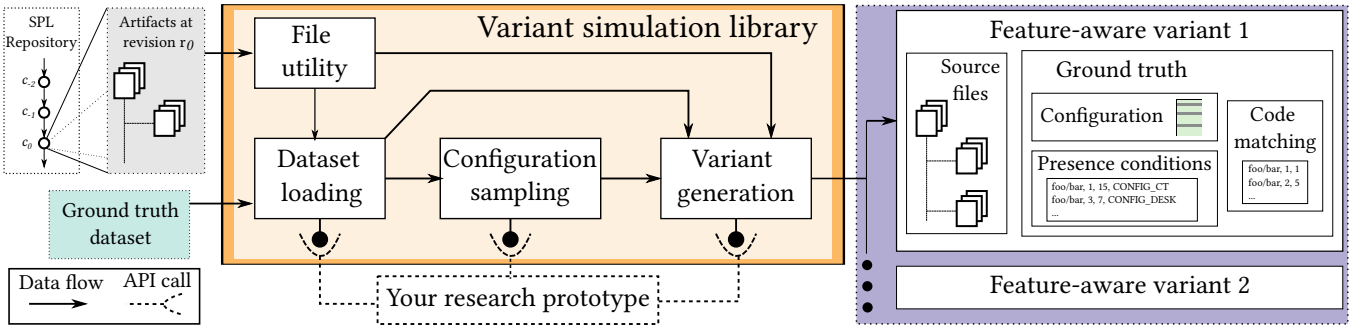

Figure 3: Overview of the variant simulation library of VEVOS.

ground truth for 5,605 of the most recent commits; more specifically all commits in the range *50239a665c88f5a9* to *b276e41835161234*, which was introduced on the 22nd of August 2010. The extraction took roughly one day for the entire history.

For Linux, running the ground truth extraction requires considerable time. We decided to validate the extraction on a subset of the commits as follows: First, we ran the extraction once for each major version (i.e., *v1*-*v5*) to broadly cover the entire history; we found that major version *v4* could be extracted successfully, while the other versions lacked the files required by *KConfigReader* and *kbuildminer*. Then, we ran the extraction for all minor versions between *v4* and *v5* (i.e. v4.0-4.20); here, the extraction succeeded for the minor versions v4.0 - v4.10. Finally, we ran the extraction once for all versions between two minor versions (*v4.4*-*v4.5*); the extraction was successful for 13,082 commits, and failed for 92 commits. The extraction for these commits required roughly one week and produced about 90GB of data. In conclusion, we estimate that it is possible to extract a ground truth for more than 140,000 commits (version v4.0 - v4.10) that span almost 2 years of the kernel's development history (2015 - 2017).

6.2 Variant Simulation

VEVOS' variant simulation library offers the loading of datasets, sampling of configurations, and generation of feature-aware variants. The loading of a dataset is straightforward and has no feasibility issues. The sampling of configurations is performed with the *FeatureIDE*-library [26]. *FeatureIDE* offers a variety of functionality for dealing with feature models, e.g., import and export of models, and random sampling of configurations for a given feature model. We were able to sample configurations for BusyBox and Linux. On average, sampling five configurations took less than 0.1*s* for BusyBox, and less than 120*s* for Linux. The generation of feature-aware variants is programming language agnostic but expects an annotation-based software product line (cf. Sec. 2), in which annotations are stored internally (i.e., within the source code), such as for the preprocessor annotations, embedded annotations [19], or feature tags [18]. On average, generating five variants took less than one second for BusyBox, and less than 120 seconds for Linux.

7 THREATS TO VALIDITY

The design and implementation of VEVOS lead to a number of threats to validity that any evaluation using VEVOS will inherit.

Internal Validity. Bugs in the implementation of the ground truth extraction or the variant simulation are a threat to the internal validity, which we mitigated by validating the correctness of the variability extraction and variant generation through testing and code reviews. Moreover, an analyzed software product line itself might contain variability bugs causing problems during the ground truth extraction, which may lead to invalid variants. It is impossible to fully mitigate this threat. Yet, cloned variants may also contain bugs, and novel methods and tools will have to account for them.

External Validity. The variants simulated by VEVOS are derived from a software product line, which leads to differences compared to cloned variants. First, instead of actually copying and adapting an existing variant as in real clone-and-own, simulated variants appear once their configuration has been derived from the feature model, which introduces a bias to the simulation of the variants' evolution. Yet, researchers may still simulate a cloning process by introducing a dedicated variant at any point in a revision history in which the variant's configuration is valid according to the feature model. Second, in real clone-and-own projects, variants might contain unintentional divergences [48] (i.e., syntactic or even semantic differences in the implementation of a common feature). In VEVOS's simulation, variants are derived from the same code base; code which is common to multiple variants is exactly the same in these variants. To date, we accept this threat to validity as it is the necessary trade-off to obtain (1) realistic datasets that are derived from real histories *and* (2) ground truth information on cloned variants.

8 CONCLUSION

We presented VEVOS, a framework that aids researchers with generating benchmarks for the empirical evaluation of clone-and-own research. Given the history of a software product line, VEVOS generates the evolution of feature-aware variants. Notably, VEVOS provides the necessary ground truth information, in form of a feature model, feature mappings, presence conditions, and code matchings, which are crucial to evaluate research on variable software systems.

ACKNOWLEDGMENTS

This work has been partially supported by the German Research Foundation (DFG) within the project *VariantSync* (TH 2387/1-1 and KE 2267/1-1), and by the German Ministry of Research and Education (BMBF) within the ITEA3 project *REVaMP* (01IS16042H).

REFERENCES

[1] Iago Abal, Jean Melo, Stefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. 2018. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *TOSEM* 26, 3, Article 10 (2018), 10:1–10:34 pages.

[2] Hadil Abukwaik, Andreas Burger, Berima Kweku Andam, and Thorsten Berger. 2018. Semi-Automated Feature Traceability with Embedded Annotations. In *ICSME*. IEEE, 529–533.

[3] Michał Antkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Thomas Schmorleiz, Ralf Lämmel, Stefan Stănciulescu, Andrzej Wąsowski, and Ina Schaefer. 2014. Flexible Product Line Engineering with a Virtual Platform. In *ICSE*. ACM, 532–535.

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

[5] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *SPLC*. Springer, 7–20.

[6] Thorsten Berger, Steven She, Rafael Lotufo, Krzysztof Czarnecki, and Andrzej Wasowski. 2010. Feature-to-Code Mapping in Two Large Product Lines. *SPLC*, 498–499.

[7] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *TSE* 39, 12 (2013), 1611–1640.

[8] Paul Maximilian Bittner, Alexander Schultheiß, Thomas Thüm, Timo Kehrer, Jeffrey M. Young, and Lukas Linsbauer. 2021. Feature Trace Recording. In *ESEC/FSE*. ACM, 1007–1020.

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

[10] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. A Robust Approach for Variability Extraction from the Linux Build System. In *SPLC*. ACM, 21–30.

[11] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature Location in Source Code: A Taxonomy and Survey. *JSEP* 25, 1 (2013), 53–95.

[12] 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 *CSMR*. IEEE, 25–34.

[13] Slawomir Duszynski, Jens Knodel, and Martin Becker. 2011. Analyzing the Source Code of Multiple Software Variants for Reuse Potential. In *WCRE*. IEEE, 303–307.

[14] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2020. Fast Static Analyses of Software Product Lines: An Example with More than 42,000 Metrics. In *VaMoS*. ACM, Article 8, 9 pages.

[15] Gabriel Ferreira, Momin Malik, Christian Kästner, Jürgen Pfeffer, and Sven Apel. 2016. Do #ifdefs Influence the Occurrence of Vulnerabilities? An Empirical Study of the Linux Kernel. In *SPLC*. ACM, 65–73.

[16] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *ICSME*. IEEE, 391–400.

[17] Tobias Heß, Chico Sundermann, and Thomas Thüm. 2021. On the Scalability of Building Binary Decision Diagrams for Current Feature Models. In *SPLC*. ACM, 131–135.

[18] Patrick Heymans, Quentin Boucher, Andreas Classen, Arnaud Bourdoux, and Laurent Demonceau. 2012. A Code Tagging Approach to Software Product Line Development. *STTT* 14 (2012), 553–566. Issue 5.

[19] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. 2015. Maintaining Feature Traceability with Embedded Annotations. In *SPLC*. ACM, 61–70.

[20] Christian Kästner. 2017. Differential testing for variational analyses: Experience from developing KConfigReader. *CoRR* (2017). arXiv:1706.09357

[21] Christian Kästner, Alexander Dreiling, and Klaus Ostermann. 2014. Variability Mining: Consistent Semiautomatic Detection of Product-Line Features. *TSE* 40, 1 (2014), 67–82.

[22] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *OOPSLA*. ACM, 805–824.

[23] Ed Keenan, Adam Czauderna, Greg Leach, Jane Cleland-Huang, Yonghee Shin, Evan Moritz, Malcom Gethers, Denys Poshyvanyk, Jonathan Maletic, Jane Huffman Hayes, et al. 2012. Tracelab: An Experimental Workbench for Equipping Researchers to Innovate, Synthesize, and Comparatively Evaluate Traceability Solutions. In *ICSE*. IEEE, 1375–1378.

[24] Timo Kehrer, Thomas Thüm, Alexander Schultheiß, and Paul Maximilian Bittner. 2021. Bridging the Gap Between Clone-and-Own and Software Product Lines. In *ICSE*. IEEE, 21–25.

[25] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. Is There a Mismatch Between Real-World Feature Models and Product-Line Research?. In *ESEC/FSE*. ACM, 291–302.

[26] Sebastian Krieter, Marcus Pinnecke, Jacob Krüger, Joshua Sprey, Christopher Sontag, Thomas Thüm, Thomas Leich, and Gunter Saake. 2017. FeatureIDE: Empowering Third-Party Developers. In *SPLC*. ACM, 42–45.

[27] Christian Kröher, Sascha El-Sharkawy, and Klaus Schmid. 2018. KernelHaven: An Experimentation Workbench for Analyzing Software Product Lines. In *ICSEC*. ACM, 73–76.

[28] Christian Kröher, Lea Gerling, and Klaus Schmid. 2018. Identifying the Intensity of Variability Changes in Software Product Line Evolution. In *SPLC*. ACM, 54–64.

[29] Jacob Krüger, Mukelabai Mukelabai, Wanzi Gu, Hui Shen, Regina Hebig, and Thorsten Berger. 2019. Where is my Feature and What is it About? A Case Study on Recovering Feature Facets. *JSS* 152 (2019), 239–253.

[30] Elias Kuiter, Sebastian Krieter, Jacob Krüger, Kai Ludwig, Thomas Leich, and Gunter Saake. 2018. PClocator: A Tool Suite to Automatically Identify Configurations for Code Locations. In *SPLC*. ACM, 284–288.

[31] Raúl Lapeña, Manuel Ballarin, and Carlos Cetina. 2016. Towards Clone-and-Own Support: Locating Relevant Methods in Legacy Products. In *SPLC*. ACM, 194–203.

[32] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2012. *Large-Scale Variability-Aware Type Checking and Dataflow Analysis*. Technical Report MIP-1212. University of Passau.

[33] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2010. Evolution of the Linux Kernel Variability Model. In *SPLC*. Springer, 136–150.

[34] Wardah Mahmood, Daniel Strueber, Thorsten Berger, Ralf Laemmel, and Mukelabai Mukelabai. 2021. Seamless Variability Management With the Virtual Platform. In *ICSE*. IEEE, 1658–1670.

[35] Jabier Martinez, Wesley K. G. Assunção, and Tewfik Ziadi. 2017. ESPLA: A Catalog of Extractive SPL Adoption Case Studies. In *SPLC*. ACM, 38–41.

[36] Jabier Martinez, Nicolas Ordoñez, Xhevahire Tërnava, Tewfik Ziadi, Jairo Aponte, Eduardo Figueiredo, and Marco Tulio Valente. 2018. Feature Location Benchmark with ArgoUML SPL. In *SPLC*. ACM, 257–263.

[37] 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 *SPLC*. ACM, 101–110.

[38] Jabier Martinez, Tewfik Ziadi, Mike Papadakis, Tegawendé F. Bissyandé, Jacques Klein, and Yves le Traon. 2018. Feature Location Benchmark for Extractive Software Product Line Adoption Research Using Realistic and Synthetic Eclipse Variants. *IST* 104 (2018), 46–59.

[39] Thilo Mende, Rainer Koschke, and Felix Beckwermert. 2009. An Evaluation of Code Similarity Identification for the Grow-and-Prune Model. *JSME* 21, 2 (2009), 143–169.

[40] Gabriela Karoline Michelon, David Obermann, Wesley K. G. Assunção, Lukas Linsbauer, Paul Grünbacher, and Alexander Egyed. 2021. Managing Systems Evolving in Space and Time: Four Challenges for Maintenance, Evolution and Composition of Variants. In *SPLC*. ACM, 75–80.

[41] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2015. Where Do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *TSE* 41, 8 (2015), 820–841.

[42] Leonardo Passos, Leopoldo Teixeira, Nicolas Dintzner, Sven Apel, Andrzej Wąsowski, Krzysztof Czarnecki, Paulo Borba, and Jianmei Guo. 2016. Coevolution of Variability Models and Related Software Artifacts. *EMSE* 21, 4 (2016).

[43] Tobias Pett, Sebastian Krieter, Tobias Runge, Thomas Thüm, Malte Lochau, and Ina Schaefer. 2021. Stability of Product-Line Sampling in Continuous Integration. In *VaMoS*. ACM, Article 18, 9 pages.

[44] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.

[45] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. 2013. Managing Cloned Variants: A Framework and Experience. In *SPLC*. ACM, 101–110.

[46] AnaB. Sánchez, Sergio Segura, JoséA. Parejo, and Antonio Ruiz-Cortés. 2015. Variability Testing in the Wild: The Drupal Case Study. *SoSyM* (2015), 1–22.

[47] Klaus Schmid, Sascha El-Sharkawy, and Christian" Kröher. 2019. *Improving Software Engineering Research Through Experimentation Workbenches*. Springer, 67–82.

[48] Thomas Schmorleiz and Ralf Lämmel. 2014. Similarity Management via History Annotation. In *SATToSE*. Dipartimento di Informatica Università degli Studi dell'Aquila, L'Aquila, Italy, 45–48.

[49] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. 2014. Capturing Variability in Space and Time with Hyper Feature Models. In *VaMoS*. ACM, Article 6, 6:1–6:8 pages.

[50] Daniel Strüber, Mukelabai Mukelabai, Jacob Krüger, Stefan Fischer, Lukas Linsbauer, Jabier Martinez, and Thorsten Berger. 2019. Facing the Truth: Benchmarking the Techniques for the Evolution of Variant-Rich Systems. In *SPLC*. ACM, 177–188.

[51] Stefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. 2015. Forked and Integrated Variants in an Open-Source Firmware Project. In *ICSME*. IEEE, 151–160.

[52] Zhenchang Xing, Yinxing Xue, and Stan Jarzabek. 2013. A large scale linux-kernel based benchmark for feature location research. In *ICSE*. IEEE, 1311–1314.

[53] Chenguang Zhu, Yi Li, Julia Rubin, and Marsha Chechik. 2017. A Dataset for Dynamic Discovery of Semantic Changes in Version Controlled Software Histories. In *MSR*. IEEE, 523–526.