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; • General and reference → Evaluation.

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

ACM Reference Format:

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. variability 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 BusyBox [17, 25, 30, 32, 43], the product-line’s feature model and
presence conditions are hidden behind the build system and pre-processor 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). Therefore, 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., #ifdef, #ifndef, 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 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 dif-
cferences, 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 ex-
licit 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 prepro-
cessor 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 prepro-
cessors such as cppp\(^1\) or coam\(^2\) 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

\(^1\)https://www.muppetlabs.com/~breadbox/software/cppp.html

\(^2\)http://coan2.sourceforge.net/index.php?page=about

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 sys-
tem 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 ver-
sion 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 pre-
processor 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 ef-
facts [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 VEVS

To address the lack of benchmarks, we propose VEVOS: a framework
for the generation of benchmarks that can be used for the evalua-
tion 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 consider-
ably 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 frame-
work [27, 47] but (1) extends it with a new plugin to analyze pres-
ence 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 con-
tainer (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 con-
sidered. 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., check-
out 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).

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., extraction³ and simulation⁴).

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 kbuild-miner [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 ground truth for 5,605 of the most recent commits; more specifically

---

³https://github.com/VariantSync/VEVOS_Extraction
⁴https://github.com/VariantSync/VEVOS_Simulation
⁵System: Intel i5-8700K @3.70GHz CPU, 32 GB DDR4 RAM; 1TB M.2 SSD
all commits in the range 50239a665e88f3a9 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 0.1s for BusyBox, and less than 120s 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 within the project VariantSync (TH 2387/1-1 and KE 2267/1-1), and by the German Ministry of Research and Education within the ITEA3 project REVaMP (01IS16042H).