Quantifying the Potential to Automate the Synchronization of Variants in Clone-and-Own

Alexander Schultheiß*, Paul Maximilian Bittner†, Thomas Thüm† and Timo Kehrer‡
*Humboldt University of Berlin, Germany, schultha@informatik.hu-berlin.de
†University of Ulm, Germany, {paul.bittner, thomas.thuem}@uni-ulm.de
‡ University of Bern, Switzerland, timo.kehrer@inf.unibe.ch

Abstract—**In clone-and-own – the predominant paradigm for developing multi-variant software systems in practice – a new variant of a software system is created by copying and adapting an existing one. While clone-and-own is flexible, it causes high maintenance effort in the long run as cloned variants evolve in parallel; certain changes, such as bug fixes, need to be propagated between variants manually. On top of the principle of cherry-picking and by collecting lightweight domain knowledge on cloned variants and software changes, a recent line of research proposes to automate such synchronization tasks when migration to a software product line is not feasible. However, it is yet unclear how far this synchronization can actually be pushed. We conduct an empirical study in which we quantify the potential to automate the synchronization of variants in clone-and-own. We simulate the variant synchronization using the history of a real-world multi-variant software system as a case study. Our results indicate that existing patching techniques propagate changes with an accuracy of up to 85%, if applied consistently from the start of a project. This can be even further improved to 93% by exploiting lightweight domain knowledge about which features are affected by a change, and which variants implement affected features. Based on our findings, we conclude that there is potential to automate the synchronization of cloned variants through existing patching techniques.**

Index Terms—**clone-and-own, change propagation, variant synchronization, version control, software product lines**

I. Introduction

Today's software is often released in multiple variants to meet varying customer requirements. While there are systematic approaches to managing variability, such as software product lines where all variants are managed using an integrated platform [1]–[3], these approaches are not feasible for all projects. Instead, developers fall back to using clone-and-own, where a new variant of a software system is created by copying and adapting an existing variant (e.g., using branching/forking capabilities of a version control system). This way, new variants are created ad-hoc and without requiring upfront investments or knowledge about variants required in the future [4]–[7]. In the long term, however, clone-and-own projects suffer from ever-increasing maintenance costs for various reasons [4]–[10]. For example, if a bug is discovered and fixed in one variant, it is often unclear which other variants are affected by the same bug and how this bug should be fixed in these variants.

Researchers started to explore the continuum between ad-hoc clone-and-own and software product lines to reduce the burden on developers [11]–[17]. The goal of such managed clone-and-own is to keep the flexibility of clone-and-own while getting rid of its maintenance problems. In this paper, we focus on a recent line of research that proposes to manage the development and maintenance of cloned variants by propagating changes of interest between them, thereby keeping the cloned variants synchronized [4], [16]–[20]. Essentially, all of these approaches rely on a change propagation facility which is based on the principle of document patching. The changes that occurred between two revisions of a source variant are represented as a series of patches which are applied to one or several target variants. On top of this basic principle, some approaches advocate to collect additional lightweight domain knowledge on cloned variants and software changes, with the goal of automating the synchronization of variants [16], [17], [21], [22]. However, the synchronization capabilities of the proposed approaches are still in their infancy and it is yet unclear how far this synchronization can actually be pushed.

We aim at quantifying the potential to automate the synchronization of variants in clone-and-own through an empirical study, simulating almost half a billion synchronization scenarios extracted from the history of a real-world multi-variant software system as a case study. Specifically, we consider variants of the software product line BusyBox. BusyBox is a highly configurable tool suite with a rich history of more than 17,000 commits and a configuration space with more than 10^{200} different variants [23]. BusyBox is widely used in practice [24]–[26] and is a common benchmark in research on multi-variant software systems [27]–[30].

Our study covers three aspects of patch-based variant synchronization. First, to gain insight on the difficulties of automated change propagation, we investigate how often propagating a change via patching succeeds or fails, depending on different levels of patch granularity (i.e., commit-, file-, and line-level patches). Second, we examine the correctness of automated patching by analyzing the outcome of each synchronization scenario. Due to the differences in the source code of variants, patches can contain desired and undesired changes, but not all patches can be applied successfully. Thus, we are interested in how often the correct outcome is achieved. Third, we investigate the potential to improve the correctness of automated synchronization when developers document lightweight domain knowledge. Specifically, we employ lightweight domain knowledge about which features

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

are affected by a change and which variants implement affected features – knowledge that is generally assumed to be available but typically undocumented in clone-and-own [9], [15]–[17], [31], [32]. This way, we can filter undesired patches in an a-priori fashion, without blindly testing their applicability. In summary, we have three main contributions:

- We present a framework that simulates the automated synchronization of variants for almost half a billion synchronization scenarios.
- We quantify the potential to automate the synchronization of variants through fully-automated patching by analyzing applicability and correctness of patches.
- We quantify to which extent automated patching benefits from utilizing lightweight domain knowledge about which features are affected by a change and which variants implement them.

II. Background and Motivation

In this section, we briefly review the state-of-the-art in engineering multi-variant software systems, with the main goal of providing the background and motivation for conducting the empirical study presented in this paper.

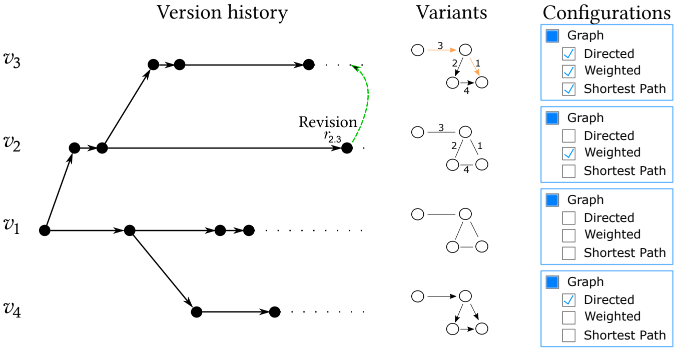
A. Software Product Lines

A *software product line* is a set of similar software variants (aka., *products*) with well-defined commonalities and variability [1]–[3], developed as a common code base (aka., integrated software platform). On the abstraction level of requirements, the commonalities and variability of a target domain are described in terms of *features*. Each variant is identified by a unique combination of features, called a *configuration* [3]. The set of valid configurations is typically specified by a *feature model* [33]. A product line is implemented by mapping the features onto implementation artifacts and choosing a variation mechanism which specifies how to generate individual variants from the common artifacts (e.g., using preprocessors, build systems, or plug-ins) [1], [3], [34]. A variant is built by selecting a desired configuration and deriving the corresponding implementation from the integrated platform.

B. Ad-Hoc Clone-and-Own

As opposed to software product-line engineering, the state-of-practice in engineering multi-variant software systems often follows a radically different pattern: The development starts with a single variant, where the overhead of product lines does not pay off. Later, further variants are added by copying and adapting an existing variant (e.g., by branching or forking), and all variants evolve in parallel; a principle which is generally known as *clone-and-own* [4]–[7].

The clone-and-own workflow is illustrated in Figure 1, using the development of a simple graph library as an example. Initially, the library comprises only a single variant, called v_1, which implements simple *undirected* graphs. Next, this variant is copied to become variant v_2, which implements the additional feature of graphs having *weighted* edges. After some modifications, variant v_2 is copied to become variant v_3. In



■ Graph
☑ Directed
☑ Weighted
☑ Shortest Path

■ Graph
□ Directed
☑ Weighted
□ Shortest Path

■ Graph
□ Directed
□ Weighted
□ Shortest Path

■ Graph
☑ Directed
□ Weighted
□ Shortest Path

Fig. 1: Motivating example: Evolution of variants in clone-and-own (optionally with configurations when managed).

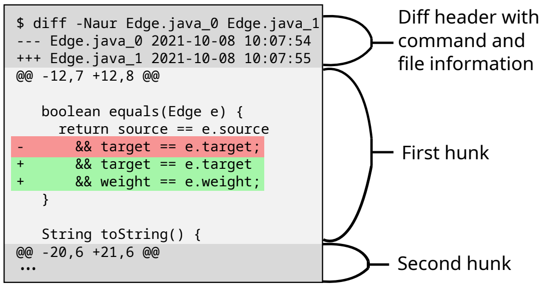
variant v_3, graphs are weighted and directed, and the graph library provides an additional utility function that calculates the *shortest path* between any two nodes. Finally, a modified version of initial variant v_1 is copied to become variant v_4, which implements directed graphs, yet without the additional features *weighted* and *shortest path*. Later on, all four variants evolve in parallel and independently of each other (e.g., due to local adjustments or bug fixes).

As illustrated in our example, clone-and-own makes it possible to create new variants in a flexible manner, yielding a family of related software products without any upfront investment. However, the flexibility of clone-and-own comes at a price. Consider, for instance, the creation of variant v_4 which shall implement directed graphs. In ad-hoc clone-and-own, this leads to redundant implementations of the feature *directed*, which has already been implemented in variant v_3. As another example, suppose that a bug has been fixed in revision $r_{2.3}$ of variant v_2. Developers will have to check manually which other variants are affected by the same bug (here, variant v_3). These are problems that could be solved by automating the synchronization of variants.

C. Basic Techniques for Synchronizing Variants

The synchronization of variants can be achieved by propagating desired changes from one variant to another. Following common terminology of version control systems, this is typically referred to as cherry-picking [35], which is technically based on the principle of document *patching*. Changes between two revisions of a *source variant* are represented as a patch (aka. (directed) delta [36], (asymmetric) difference [37], or edit script [38]), which is applied to a *target variant*.

Technically, patching is realized by providing two operators, commonly referred to as *diff* and *patch*. These operators rely on a common representation of the underlying documents which are to be processed, and on a set of change operations that can be used to modify these documents. Language-specific implementations are tailored to the language's (abstract) syntax, such as structural diff and patch operators that work on the abstract syntax tree (AST) of source code. In contrast, language-agnostic implementations of diff and patch rely on a generic representation of documents, a prominent example of this being



```
$ diff -Naur Edge.java_0 Edge.java_1
--- Edge.java_0 2021-10-08 10:07:54
+++ Edge.java_1 2021-10-08 10:07:55
@@ -12,7 +12,8 @@

   boolean equals(Edge e) {
     return source == e.source
-      && target == e.target;
+      && target == e.target
+      && weight == e.weight;
   }

   String toString() {
@@ -20,6 +21,6 @@
 ...
```

Fig. 2: Example of a patch with three changes in the first hunk.

```
11 ...
12
13 boolean equals(Edge e) {
14   return source == e.source
15     && label  == e.label
16     && target == e.target;
17 }
18
19 String getLabel() {
20 ...
```

Listing 1: Propagating the change from Figure 2 to this variant of the Edge class featuring *labeled* edges fails due to a different context (i.e., Line 3).

the Unix utilities diff and patch [39] which process textual documents in a line-based manner.

In our study, we use standard and language-agnostic diff and patch operators as they are widely used in practice and serve as the basis for implementing cherry-picking in version control systems such as Git or SVN. A patch comprising the changes to be propagated is created by diffing the original and the changed version of a document. An example of a patch is shown in Figure 2, where the method equals of the class Edge from the graph library of Figure 1 is extended to also consider *weights*. A patch basically consists of a set of so-called *hunks* [39]. A hunk describes which changes (i.e., insertions and deletions of lines) should be made to a certain text block. Context lines are added around each hunk as shown in Figure 2 where the changed lines (green and red) are surrounded by additional lines. A patch operator takes a patch as input and applies the specified changes to a given target document, producing a patched version. The patch operator exploits the context lines when searching for the correct location in the target file to apply a hunk. Starting from the specified line number (cf. Figure 2), the operator searches for a matching context in the target file and applies the patch to the first matching location. If no match is found, the application fails and the hunk is written to a reject file.

Notably, cherry-picking based on the principle of patching is not to be confused with merging [40]. With cherry-picking, selected changes from selected commits of one branch are applied to another branch. Merging aims at integrating concurrent modifications into one unified version. All changes from all new commits of one branch are applied to the another branch. Thus, merging is inadequate for synchronizing co-evolving variants; these are supposed to *have* differences, whereas merging tries to get rid of them.

D. Towards Automated Synchronization

Although cherry-picking may serve as a technical basis for synchronizing cloned variants, its manual application is tedious and prone to errors [18]. Developers have to manually determine which changes should be propagated to which variants. For each potential target variant, developers have to perform a cherry-pick. Furthermore, the propagation of changes may technically fail. Since the code bases of cloned variants exhibit differences, the context of a patch might not match the desired location in a target variant. For example, applying the change from Figure 2 to a variant with directed, weighted, and labeled edges, as shown in Listing 1, will fail because of a non-matching context: The context-line 15 in Listing 1 does not exist in the patch.

In this paper, we quantify the potential for overcoming these problems to automate change synchronization. First, we study how far an automated synchronization is feasible in real-world clone-and-own where the necessary domain knowledge of which changes to apply in which target variants is usually available but undocumented. Second, we investigate to which degree an automated synchronization is improved when such lightweight domain knowledge would be documented by developers in terms of features and configurations.

III. Study Design

Our study's research goal is to quantify the potential to automate the synchronization of variants for which we define three research questions. RQ1 quantifies the automation potential in terms of the general applicability of patches (i.e., rate of application without failure). RQ2 quantifies the potential in terms of how often patching leads to the correct result. RQ3 explores to which extent explicit domain knowledge could improve the automation potential.

RQ1 *(Applicability): Which fraction of changes in a variant can be propagated blindly to other variants through automated patching?* There are a number of scenarios where trying to apply a patch to another variant may fail. Such failures are likely to occur when patching software variants, as variants expose differences in their code base. Patch application failures could decrease the potential of automating variant synchronization. Thus, we want to quantify how many patches can be applied.

Furthermore, the applicability of a patch might also be affected by its granularity which in turn depends on the number of changes that happened between two versions of the variant from which the patch was created. We account for the size of patches at three levels of granularity: *commit-level* patches that contain all changes that were made in a commit, *file-level* patches that contain all changes made to a specific file in a commit, and *line-level* patches that contain the change of exactly one line.

RQ2 *(Correctness): How often does blindly propagating changes to other variants lead to the expected patch result?*

When quantifying the applicability of a patch (RQ1), we apply patches blindly to variants without considering whether a patch contains changes that are desired or undesired in a target variant. A change is desired in a variant if it affects features that are implemented by this variant, and undesired if it affects features that are *not* implemented by it. Thus, we quantify how often blindly propagating changes through automated patching leads to the expected patch result, i.e., all desired but no undesired changes are propagated.

RQ3 *(Domain Knowledge): To what extent can lightweight domain knowledge improve the correctness of change propagation?* With RQ1 and RQ2, we observe variant synchronization without domain knowledge – all changes are blindly propagated to other variants. Both research questions are relevant for immediately automating variant synchronization in practice, as they only depend on existing tools without additional prerequisites. With RQ3, we want to explore to which extent utilizing lightweight domain knowledge could further improve the potential to automate variant synchronization. We focus on domain knowledge about the configurations of variants and the features that are affected by a code change, as this knowledge is generally assumed to be available [9], [15], [31], [32]. By combining this knowledge, it should be possible to determine whether a change is desired or undesired in a specific target variant. We hypothesize that we can utilize this knowledge to filter out all undesired changes (i.e., changes that must not be applied), which in turn should improve the correctness of patching. Please note that even if lightweight domain knowledge is available in a project, we cannot assume that it is explicitly documented, which means that synchronization techniques cannot utilize this knowledge immediately in a real clone-and-own project. Thus, with RQ3, we also quantify whether making this knowledge explicit is worth the required effort and cost.

A. Experimental Subject

Requirements for Simulation of Clone-and-Own: Blind change propagation (RQ1 and RQ2) can be applied to any software system. Thus, the most basic requirement is that the subject comprises the evolution history for a set of variants on which we can simulate patch-based change propagation. Change propagation with lightweight domain knowledge (RQ3) requires domain knowledge about the configurations of variants and the features affected by a change.

Requirements for Evaluation: We can determine the applicability of patches (RQ1) without additional information. Besides applicability, we also need to determine the correctness of a patch result (RQ2 and RQ3), for which we have to know which changes are desired or undesired in a target variant, and whether a change has been applied to the correct location. Desired and undesired changes can be determined by comparing the affected features with the configuration of the target variant. Whether a change has been applied to the correct location requires knowledge about where this patch should have been applied.

Subject Selection: We found no clone-and-own project that met the requirements of our study. Thus, we consider variants stemming from a software product line, which is a common strategy for obtaining an experimental subject for clone-and-own research [15], [32], [41], [42]. Specifically, we selected the software product line BusyBox, a tool suite that contains standard Unix tools, written in C, for resource-constrained systems. BusyBox fulfills all requirements specified at the start of this section. It has a rich history of more than 17 thousand commits. BusyBox is widely used in practice [24], [26], [43] and for research on multi-variant software systems [27]–[29], [44]. Furthermore, BusyBox is highly configurable and comprises families of program variants, for instance, to meet platform-specific requirements. BusyBox' variability is implemented with C preprocessor directives (`#ifdef`) and *Kbuild*,[1] which we can analyze to extract the required domain knowledge.

B. Extraction of Required Domain Knowledge

We extract the required domain knowledge from BusyBox with VEVOS's ground truth extraction module [45], which analyzes the *Kbuild* and source files for each commit in BusyBox' history with the help of *KernelHaven* [46], a plugin-based framework for static product line analysis. For BusyBox' analysis, we configured VEVOS to use *KernelHaven*'s *KConfigReader*, *kbuildminer*, and *CodeBlock* plugins. *KConfigReader* [47], analyzes *Kbuild*'s configuration files and determines constraints among features, thereby yielding a feature model needed for sampling variant configurations. *Kbuildminer* [48], analyzes *Kbuild*'s makefiles with respect to the feature model, in order to determine the presence conditions of files. *CodeBlock* [46] parses the source code files and analyzes the contained preprocessor directives yielding presence conditions for each line in a file. The presence condition of a line of code determines in which variants the line should be included. From presence conditions of changed lines, we can derive which features are affected by a change, and to which variants a change should be propagated.

Note that VEVOS' domain knowledge extraction is only needed for obtaining a ground truth for evaluation, and the lightweight domain knowledge required in RQ3. In practice, this lightweight domain knowledge could be attained with feature trace recording [17]. The synchronization without domain knowledge (RQ1 and RQ2) uses existing patching techniques and could already be applied in any software project without extracting domain knowledge.

C. Simulation of Automated Variant Synchronization

Once the required domain knowledge is extracted from BusyBox, we begin the simulation of automated variant synchronization as presented in Figure 3. In each iteration,

[1] *Kbuild* is a build system originally designed for the Linux kernel that manages variability statically through makefiles that control the build process (i.e., conditional compilation), and configuration files that define the active features. Depending on the active features, the makefiles prepare the environment and call the compiler to compile the source code files which can also contain variability in form of preprocessor directives.

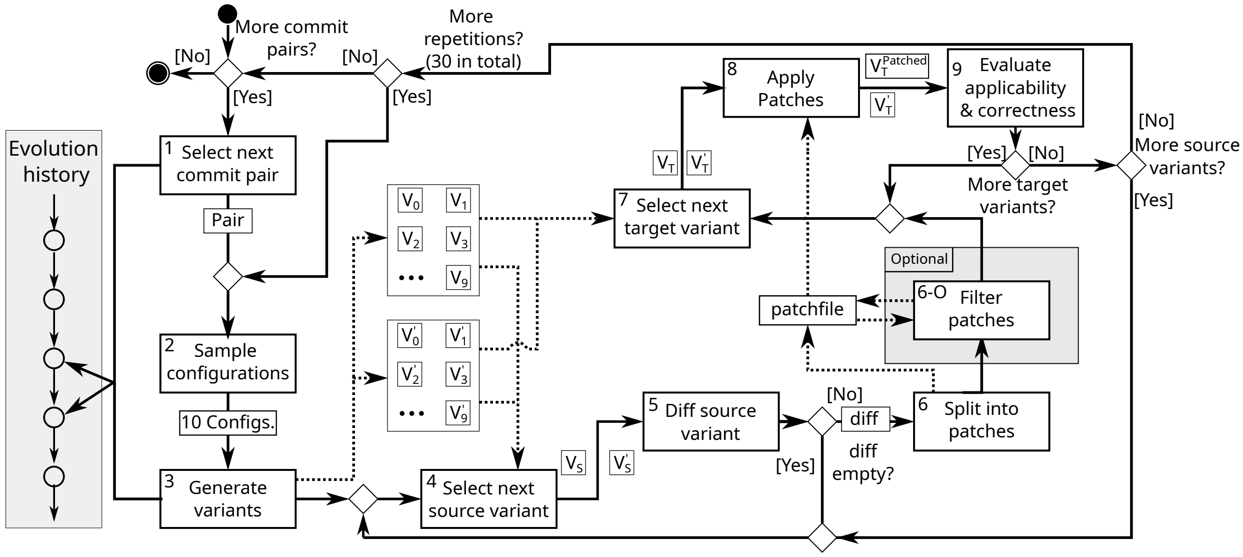


Fig. 3: Simulation of automated change synchronization.

we consider a pair of consecutive commits (1), which together represent one step in the evolution of BusyBox. For each commit pair, the simulation consists of two steps: First, the preparation phase in which a set of variants is generated (cf. Section III-C1); Second, the change propagation phase in which diffing and patching are applied to simulate automated synchronization (cf. Section III-C2). Thereby, we simulate a scenario in which all variants are synchronous until new changes are introduced to one of the variants. Once the changes are committed, they are immediately propagated to the other variants. This corresponds to applying automated change propagation from the start of a clone-and-own project.

1) Sampling and Generation of Variants: Before we can generate a variant, we need its configuration in order to determine which code its source files should contain. BusyBox 1.18.0 comprises 854 features inducing more than 10^{200} valid configurations [23]. Hence, considering all possible variants is not feasible and clone-and-own projects are far smaller with usually about ten variants [5] (cf. paragraph VI-0b). Therefore, we sample 10 random configurations (Step 2 in Figure 3). We sample variants based on the union of the two feature model versions using VEVOS [45] which internally calls the *FeatureIDE* library [49], which in turn contains functionality for analyzing and manipulating feature models. By analyzing the constraints that may exist between features, *FeatureIDE* can randomly sample valid configurations for a given feature model.[2] We mitigate the bias of random sampling by repeating the simulation 30 times (which was feasible with respect to total simulation time) for a commit pair with new variants in each repetition.

Once a set of configurations is sampled, we generate variants by generating their source code files for both commits (Step 3) with VEVOS' simulation module [45]. The generation is done by comparing a configuration with the presence condition of each line in the original BusyBox sources, as done by the pre-processor directives when BusyBox is built with *Kbuild*. If a presence condition is fulfilled according to the active features in the configuration, the line is copied to the variant's source code, otherwise it is discarded.

2) Simulation of Variant Synchronization with `diff` *and* `patch`*:* After we generated variants, we start the simulation of patch-based change synchronization. For exercising diffing and patching, we use Unix `diff` and `patch`, standard tools being in use for decades (cf. Section II-C). We configure `diff` as recommended in the *Notes for Patch Senders* section of `patch`'s documentation (i.e., `diff -Naur old new`). In this configuration, `diff` treats absent files as empty, all files as text, considers all files in a directory recursively, and collects three lines of unified context. We use `patch` in its default configuration with the *forward* option that prevents `patch` from checking for duplicate applications, which cannot occur in our controlled simulation. With these configurations, renamed files are treated as deleted and inserted, and files that are created in the source variant are also created in the target.

We repeat the simulation (4-9) for all possible source-target combinations in a set of variants (cf. Figure 3). First, we select the next source variant V_S (i.e., the variant from which the patch is created) together with its next version $V_S^{'}$ in BusyBox' commit history (Step 4). Then, we apply `diff` to the two versions of the source variant yielding the difference between both versions (Step 5). Next, we split the calculated difference into line-level patches that are written to a patch file (Step 6). We consider line-level patches as they are the finest granularity; thus, they can be combined to represent file-level and commit-level patches. Afterwards, we select the next target

[2] For sampling, *FeatureIDE* uses the Sat4J SAT solver [50]. *FeatureIDE*'s developers changed the solver's variable assignment heuristic for random sampling: Each variable is randomly set to true or false, and the order in which variables are set is random. However, configurations still depend on the rest of the solver's architecture and are therefore not uniformly distributed.

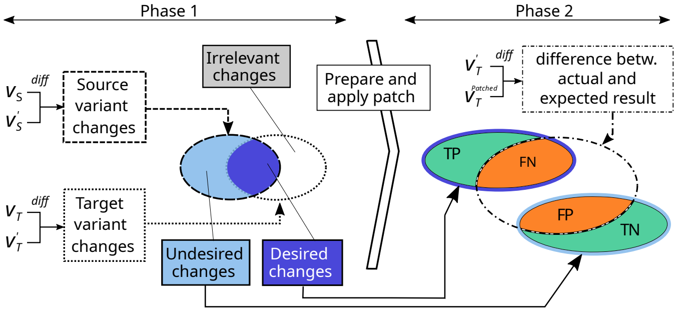


Fig. 4: Evaluation of patch results.

variant V_T (Step 7) to which we apply each line-level patch by calling `patch` with the patch file as argument (Step 8). Once the patches are applied, we evaluate the applicability and correctness of the patches (cf. Section III-D) (Step 9), and continue with the next target variant (Step 7). If there are no more target variants left, we consider the next source variant (Step 4). After all combinations have been considered, a new set of random variants is sampled and generated (Steps 2-3). We repeat the entire process (Steps 1-9) 30 times for each commit pair to cover a broad spectrum of the possible configurations.

For RQ3, we additionally filter out all changes that should not be propagated. A change is filtered if the features affected by it are not implemented in the target variant. Therefore, Step 6 can be followed by the additional filtering of line-level patches as an optional step (6-O).

D. Evaluation Metrics

We measure the applicability of patches by counting the number of patches that are applied by `patch` without failure. Unix `patch` logs all failed patches and writes failed hunks (see Section II-C) to a file which we then parse to determine failed patches. We consider a patch as failed if at least one of its changes is not propagated to the target variant.

We measure the correctness of patches in two phases which are shown in Figure 4. In the first phase (left), we determine the sets of desired changes (i.e., should be propagated) and undesired changes (i.e., must not be propagated). The phase starts with the retrieval of two change sets by applying `diff` to the two versions of the source variant and to the two versions of the target variant. The calculated change sets are (1) the changes in the evolution of the source variant (i.e., the changes in all considered patches), and (2) the changes in the evolution of the target variant (i.e., the changes that are expected). The intersection of both sets determines the desired changes, while changes that occurred only in the source variant are undesired. Changes that occurred only in the target variant are not of interest for our synchronization scenario as they did not occur in the source variant and thus cannot be propagated (i.e., irrelevant changes). Rather, they model parallel development whose behavior is out of the scope of this study.

The second phase of the evaluation (right of Figure 4) starts once all patches have been applied. To measure the correctness of patching, we observe the difference between the target variant after patching $V_T^{Patched}$ (i.e., the actual result) and the target variant in its next evolution stage $V_T^{'}$ (i.e., the expected result). By diffing $V_T^{Patched}$ and $V_T^{'}$ we obtain a set of all changes that have to be applied to complete the evolution of the patched target variant $V_T^{Patched}$ to become $V_T^{'}$. A change in this set is either a desired change that was not applied, an undesired change that was applied, or a change that only occurs in the evolution of the target (i.e., an irrelevant change). By comparing the obtained sets, we can calculate true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN).

Desired changes that do *not* appear in the observed difference are counted as true positives (TP) as these changes were correctly applied. Undesired changes that are part of the observed difference are counted as false positives (FP) as these changes were applied but should not have been. Undesired changes that do not appear in the observed difference are counted as true negatives (TN) as these changes should not be applied and were not. Finally, desired changes that appear in the observed difference are counted as false negatives (FN) as these changes should have been applied but were not.

The remaining differences that were not classified as TP, FP, TN, or FN, are irrelevant changes and can therefore be ignored as described above. Once TPs, FPs, TNs, and FNs are determined, we calculate precision, recall and balanced accuracy [51]. We choose balanced accuracy over normal accuracy because it mitigates the bias of unbalanced data. We have to account for unbalanced data, because the sets of applied and failed patches, as well as the sets of desired and undesired changes can be of arbitrary size.

For RQ3, we repeat our simulation with domain knowledge but we require no additional metrics as we evaluate the correctness of patch results as we do for RQ2. Hence, we have gathered all the data, tools, and metrics that are necessary to run and evaluate our simulation (cf. Section III-C).

IV. Quantification of Automation Potential

We implemented the simulation of automating the synchronization of variants generated from BusyBox as described in Section III-C2 in Java.[3] Our extraction of the required domain knowledge processes all commits in the history of BusyBox, which comprised a total of 17,711 commits when we ran the variability extraction (newest commit: *83e20cb81ca6d22a*). We were able to extract domain knowledge for the 5,605 most recent commits (oldest commit: *b276e41835161234*). We could not consider older commits, because these commits did not contain all *Kbuild* files required by *KernelHaven*. Nevertheless, the commits considered in our study comprise more than 10 years of development (Aug 2010 - Sep 2021).

We simulate the synchronization of variants for these 5,605 commits of BusyBox. In total, the simulation processes 9,667,506 commit-level patches, 17,706,045 file-level patches, and 498,359,460 line-level patches.

[3] The full replication package can be found on Zenodo [52] and GitHub: https://github.com/VariantSync/SynchronizationStudy

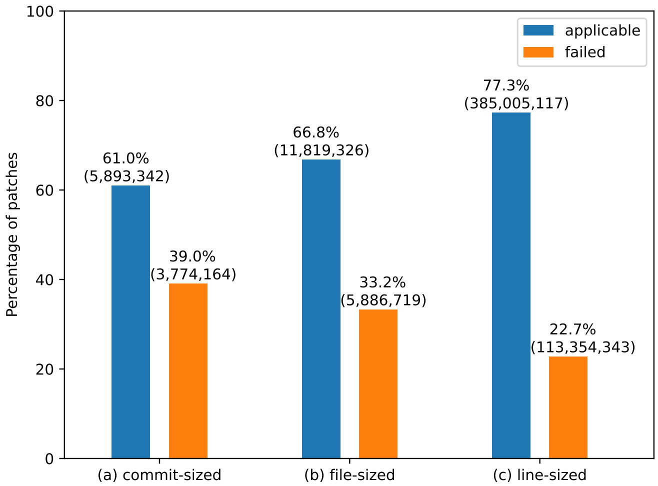


Fig. 5: Applicability of patches depending on granularity. (RQ1)

A. *(RQ1) Applicability of Blindly Propagated Changes*

Figure 5 presents the applicability of commit-level, file-level, and line-level patches. Failures occur when the context of a patch cannot be found due to differences in the code base of source and target variant as explained in Section II-C. Interestingly, most patches are applicable though.

The applicability of commit-level patches depends on the distribution of changes across commits. For instance, if each commit in the history contains at least one change that cannot be applied, the applicability of commit-level patches would be zero percent. Vice versa, if only one commit contains all changes that cannot be applied, the applicability would be almost perfect. Thus, our results show, that *most* commits (61.0%) contain *only* changes that can be applied without failure. This is also the case for the applicability of file-level patches, as *most* changed files (66.8%) contain *only* changes that can be applied without failure. Finally, the applicability of line-level patches is equivalent to how many changes are applied as each line-level patch comprises exactly one change in terms of text-based diffing. 77.3% of all line-level patches and thus changes can be applied without failure.

We believe that the high applicability of patches is a first indicator that there is potential to automate the synchronization of variants through diffing and patching. If most commit-level patches can be applied automatically, developers have to deal with failed patches less frequently. Moreover, based on our own experience, we suppose that it is easier for a developer to deal with failed patches, if only some of the changed files lead to failed patches.

Patches fail due to differences in the code base of variants but 61.0% of commit-level, 66.8% of file-level patches, and 77.3% of line-level patches are applicable. Most commits to a variant can be propagated without manual effort. Moreover, most changed files contain only applicable changes, which should reduce the effort to manually handle failures.

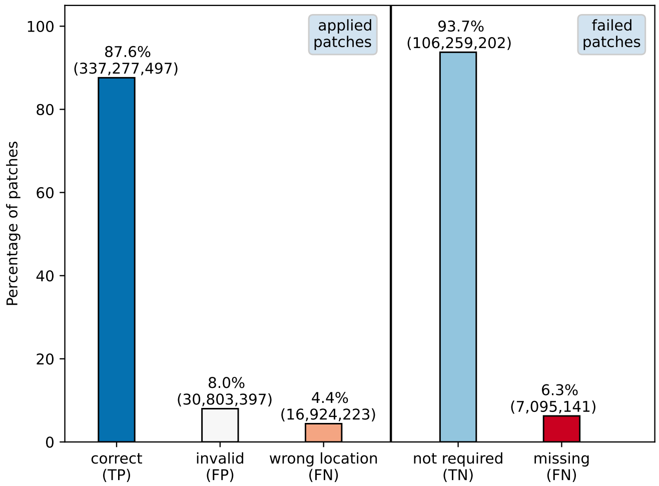


Fig. 6: Classification of results for applicable and failed patches. The applicable patches (left) and the failed patches (right) correspond to the patches in Figure 5 (c). (RQ2)

B. *(RQ2) Correctness of Blindly Propagated Changes*

Some of the applicable patches might contain undesired changes, and not all failed patches are desired. Thus, to determine the correctness of blindly propagating changes through patching, we investigate the outcome of applicable and failed line-level patches. Figure 6 presents a breakdown of applicable and failed patches into five types of patch outcomes, which we observe during the evaluation of results. The five types fall into TP, FN, FP, and TN (cf. Section III-D):

- *correct* (TP): desired changes that were applied to the correct location by an applicable patch.
- *wrong location* (FN): desired changes that were applied to the wrong location by an applicable patch.
- *invalid* (FP): undesired changes that were applied.
- *missing* (FN): desired changes that were not applied due to technical failure.
- *not required* (TN): undesired changes that were not applied due to technical failure.

Of the 385,005,117 applicable line-level patches (left side in Figure 6) 87.6% (337,277,497) are applied correctly (TP), 8.0% (30,803,397) are invalid (FP), and 4.4% (16,924,223) are applied to the wrong location (FN). At the same time, most of the failed patches (right side in Figure 6) are actually undesired changes. Of the 113,354,346 failed patches 93.7% (106,259,202) are undesired, and 6.3% (7,095,141) are missing patches that were desired.

This distribution reveals that most applicable patches (92%) contain desired changes, and that almost all desired changes that are applied, are applied to the correct location in the source code. For desired changes that were applied to the wrong location, we cannot say with certainty whether they were applied correctly or not. Even if the changes were not applied to exactly the same location as seen in the evolution of the target variant (syntactic difference), the target variant's code

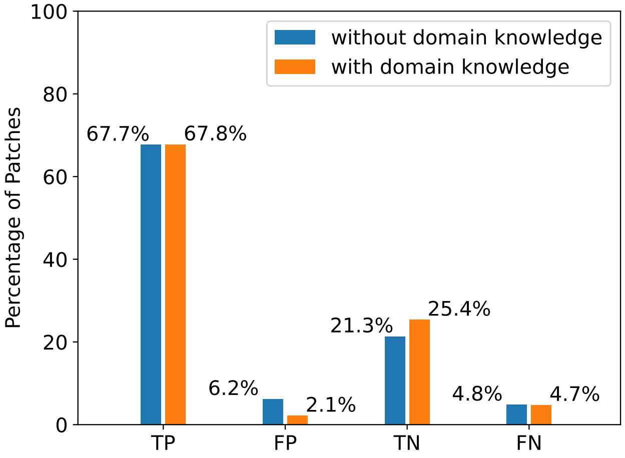


Fig. 7: Patch results without and with domain knowledge. (RQ2+3)

might still function as intended (semantic equivalence). We apply a conservative classification of such changes as incorrect (i.e., counting them as false negatives) to mitigate the bias caused by desired patches being applied incorrectly.

Table I shows the resulting values for precision, recall, and balanced accuracy. For now, we concentrate on the results shown in the first row, leaving the second row for our discussion of RQ3. The fully-automated application of `diff` and `patch` for variant synchronization achieves a precision of 0.92, recall of 0.93, and balanced accuracy of 0.85. The high precision shows that almost all applied patches were applied correctly, and the high recall that it was possible to propagate most desired changes without failure.

Our quantification of applicability and correctness shows that simple text-based diffing and patching provides a solid foundation for the automation of variant synchronization. Almost all propagated changes are also desired, and almost all desired patches can be applied. Thus, a majority of patches yield the correct result (balanced accuracy: 0.85).

C. (RQ3) Utilizing Domain Knowledge

In RQ3, we hypothesized that it should be possible to filter out undesired changes when lightweight domain knowledge is available. As a result, the correctness of automated patching should improve. Figure 7 presents the percentages of TPs, FPs, TNs, and FNs of patching blindly (RQ2) against patching with lightweight domain knowledge (RQ3). Our results presented in the second row of Table I show that a majority of undesired changes can be filtered. We are able to reduce the number of false positives by more than 20 million which is a reduction of 65%; 10,672,027 false positives remain.

By inspecting false positives, we found that they occur when the code which is to be propagated by a patch already exists in the target variant. In our simulation, which considers variants generated from BusyBox, this scenario can occur if the presence condition of code in BusyBox changes. Changing the presence condition alters the set of variants that contain the code. As a result, code can be added to a variant that is already present in other variants; the addition is still propagated to all variants that fulfill the presence condition and variants can receive code which they already contain.

This is a realistic scenario in a clone-and-own project. Imagine a synchronization mechanism is already in place in such a project, and a developer decides that a feature implemented in other variants is required in the variant they are working on; thus, they copy the code implementing this feature to their variant manually. In this case, the synchronization mechanism could classify this code as newly added and could try to propagate it back to the variant where it was copied from. This is a problem that cannot be solved with only lightweight domain knowledge about the features that are affected by a change, and knowledge of the configurations.

Besides the reduction of false positives, we also observe a slight increase in the number of true positives. We believe that this improvement is due to the reduced number of undesired changes being propagated, thus, the likelihood of propagating a desired change correctly increases.

Patching with lightweight domain knowledge presents an improvement over patching blindly, as it is possible to filter out the majority of undesired changes. While it is not possible to filter *all* undesired changes, precision still increases from 0.92 to 0.97, and balanced accuracy from 0.85 to 0.93.

D. Discussion

We find that the simple mechanisms of diffing and patching exhibit potential to automate the synchronization of variants, if the automation is used consistently from the start of a project. In this setting, even blindly propagating patches to other variants achieves high precision and recall of above 90%. This suggests that the very applicability of a patch could be a useful indicator to estimate if that patch should be propagated to another variant in case domain knowledge is lost or otherwise unavailable. We find that most commits to a variant can be propagated to other variants without failure. Thus, developers do not have to frequently resolve failed patches. Furthermore, if failures occur, they only concern some of the changed files, which should reduce the difficulty of resolving the failures.

With respect to precision and recall, we believe that improving the precision of automated patching is considerably more important than improving the (high) recall. *Why?* The majority of false negatives is the result of failed patches that are explicitly documented during the automated patch application and can therefore be easily investigated by developers. In contrast, false positives are the result of undesired changes having been applied *silently*. Thus, a developer would have to analyze the code after patching. This fuels distrust in any synchronization tool with insufficient precision, as developers can never be sure no corruption occurred and always have to perform a manual check.

Finally, we found that gathering lightweight domain knowledge in form of configurations and the features affected by a

TABLE I: Precision, recall, and balanced accuracy of patching with and without domain knowledge.

Research question	Patching with domain knowledge?	TP	FP	TN	FN	Precision	Recall	Balanced Accuracy
RQ2	✗	337,277,497	30,803,397	106,259,202	24,019,364	0.92	0.93	0.85
RQ3	✓	337,681,325	10,672,027	126,390,572	23,615,536	0.97	0.93	0.93

change improves the accuracy of automation from 85% to 93%, as undesired changes can be filtered. We suppose, that this improvement is worth the cost of documenting this knowledge explicitly. Investigating the remaining false positives revealed that tools with the aim of automating variant synchronization have to account for manual copying of code between variants, which poses an interesting challenge for future work. To solve this problem, more domain knowledge (e.g., the presence conditions for all code) might be required. Thus, variant synchronization tools should focus on retrieving or explicitly documenting domain knowledge.

V. Threats to Validity

a) Construct Validity: We assess the correctness of a patch based on the difference between the expected outcome and the observed outcome. Thus, we do not account for dependencies between applicable and failed patches. For instance, a number of patches that were applied correctly (true positives) might add code that depends on code changed by patches that were not applied (false negatives). However, there is a clear indication of a patch failing; developers will know that changes are missing and can fix broken dependencies. Thus, we decided that considering only the difference between observed and expected outcome is the best approach.

Our study design does not explicitly differentiate between changes to source code that is the same across all variants and source code that has differences (i.e., variability). Our results might be biased by the degree of variability in the variants, as only changes to source files with variability can lead to *invalid* or *missing* changes (cf. Section IV-B). However, we are interested in the automation potential, regardless of whether source code is common or variable. Moreover, we argue that BusyBox inherently comprises more variability than clone-and-own projects, because BusyBox comprises hundreds of features inducing huge numbers of valid configurations (i.e., 854 features and 10^{200} configurations in version 1.18.0 [23]). We repeat the simulation 30 times with different variants for each commit; hence, we cover considerably more possible configurations (i.e., up to 300 for each commit) than can be expected in a typical clone-and-own project.

The configuration of `diff` and `patch` (cf. Section III-C) might impact the outcome of the patch application. We have not tried to optimize the configuration to yield the best possible results, but instead simply chose the configuration of `diff` recommended in `patch`'s official documentation, and the default configuration of `patch`. We believe that this is the best approach, because (a) the default configurations are likely the ones that are commonly used in practice, and (b) any optimization might result in overfitting to variants of BusyBox, which would threaten the generalizability of our results.

b) Internal Validity: To extract domain knowledge, we use `VEVOS` which uses other third party tools (i.e., *KernelHaven*, *KConfigReader*, *kbuildminer*, *CodeBlock*). `VEVOS` and the other tools might have bugs that result in incorrect or only partial domain knowledge being extracted, and BusyBox undergoes additional preprocessing by *KernelHaven*. We generate variants based on this domain knowledge and the preprocessed source code. This might lead to a bias in our study, as the variants might differ from BusyBox' actual variants (i.e., variants generated based on correct and complete domain knowledge without preprocessing). However, extracting domain knowledge from BusyBox with *KernelHaven* is the best possible option, as we found no other tools that provide the required data in an accessible format. Furthermore, the variants are generated based on the extracted domain knowledge that is also used to evaluate the outcome of patching. Hence, there should be no discrepancies in the evaluation. Furthermore, we manually sampled and validated the extracted domain knowledge of several dozen commits.

Potential bugs in the implementation of our simulation (cf. Figure 3) pose another threat to internal validity. To mitigate this threat, we applied extensive manual and automated testing and code reviews to validate the correctness of our tooling. Moreover, we manually inspected patch results during testing to confirm that they are evaluated as intended.

Finally, we randomly select variants through non-uniform random sampling with *FeatureIDE*. This might bias our results, because random variants might not be representative, and the same variant might be sampled more than once. However, we only sample *valid* variants from BusyBox' huge configuration space, for which it is unlikely that the same variant is sampled twice. To further mitigate the bias of random variants, we repeat the simulation 30 times for each commit while considering 10 variants at a time.

c) External Validity: Considering just a single subject system threatens the external validity of our results. However, BusyBox has an extensive history with over 5,000 commits for which the required domain knowledge could be extracted. In total, our quantification is based on the simulation of almost half a billion patch applications. Thereby, we cover a broad spectrum of possible synchronization scenarios.

BusyBox is a software product line and not a clone-and-own system. Thus, the variants of BusyBox do not expose unintentional divergence by construction [19], [53], [54]: Code common to multiple variants is always exactly the same in all variants. However, we consider a scenario in which the synchronization of variants is automated from the start of a project; variants are synchronized as soon as new changes are committed. In this scenario, unintentional divergences only occur, if developers do not handle incorrect or missing changes during automated patch application. We found no suitable approach to simulate unintentional divergences without introducing additional threats to validity.

As BusyBox is a software product line, its history does not reflect concurrent changes to variants that could lead to conflicts during change propagation; any conflict reduces the applicability. This is a potential bias in our results, because concurrent changes are possible in real clone-and-own projects. However, we are not aware of any suitable simulation strategy for concurrent changes. Depending on a chosen strategy, the impact of concurrent modification could range from no changes having a conflict (i.e., observed applicability) to all changes having a conflict (i.e., zero applicability). We concluded that any bias of simulating concurrent changes is greater than the bias of not simulating them. Moreover, conflicts may occur in any project and we consider solving them to be orthogonal to solving the problems that are typically faced when synchronizing variants (cf. Section II-D).

VI. Related Work

While general approaches of how to transition from ad-hoc to managed clone-and-own have been already considered in Section II, we review related work that reports about empirical studies on change propagation and clone-and-own.

a) Empirical Studies on Change Propagation: Recent research on managed clone-and-own has suggested to synchronize co-evolving variants using a dedicated change propagation operator based on the principle of document patching [4], [16]–[20]. However, the proposed operators have been only described conceptually [4], [16], [17], or they have been implemented as early research prototypes whose applicability is yet limited to academic examples [18]–[20].

With goals and assumptions different from ours, change propagation has been extensively studied in the context of single-variant systems [55]–[59]. Here, change propagation refers to the maintenance task of propagating evolutionary changes (e.g., to an interface) to other parts of the same system (e.g., all clients using that interface). Both the problem itself as well as envisioned solutions are fundamentally different from our context of managed clone-and-own.

b) Empirical Studies on Clone-and-Own: Cloning in the large has been investigated in an empirical study by Dubinsky et al. [5]. They investigate the cloning culture in six industrial clone-and-own. Their main goal is to identify the perceived advantages and disadvantages of clone-and-own. In particular, they identified several practical benefits of clone-and-own from an organizational point of view, notably simplicity, availability, and independence of developers. These benefits have been confirmed by other experience reports and exploratory studies on clone-and-own in practice [60], [61]. However, none of these studies has put an emphasis on investigating the synchronization of cloned variants through change propagation.

Cloning as a small-scale phenomenon in single-variant systems has been studied in the context of clone management [62], [63]. Interestingly, similar to recognizing the benefits of clone-and-own in the large, several researchers provide empirical evidence that the goal of reaching redundancy-freedom by eliminating code clones is not always desirable [64]–[67]. While their results are paving interesting research directions on how to manage code clones in the small, they do not provide any insights on how to manage cloned variants.

Studies that, like ours, simulate the evolution of a multi-variant system are widespread within the research in this field. In particular, methodologies and techniques supporting clone-and-own have been evaluated by generating a set of variants from an existing software product line [15], [32], [41], [42], [68]–[71], or by using existing clones and their revision history as experimental subjects [15], [21], [72], [73]. All of these studies have evaluated techniques that support the migration of cloned variants into an integrated platform, with a specific emphasis on the preparatory steps of variability mining [15], [32], [69], [70], [73] and feature location [21], [71]. However, as argued in [16], migrating a set of cloned variants into product line is out of the scope of our envisioned paradigm of supporting clone-and-own, where cloned variants are supposed to co-exist and synchronized through change propagation.

VII. Conclusion

In this work, we quantified the potential of propagating changes between cloned variants in a clone-and-own software system automatically. Therefore, we empirically inspected almost half a billion patch scenarios derived from a large-scale real-world system, namely BusyBox.

We found that the majority of patches is applicable automatically, even when propagating changes blindly across variants. In fact, the very applicability of patches proves to be a useful indicator for determining if a target variant should actually receive a patch or not. Moreover, blind patching produces correct results in the majority of cases with an accuracy of 85% and precision of 92%.

We confirmed the hypothesis of a recent line of research that gathering lightweight domain knowledge in form of configurations (i.e., knowing the features implemented in each variant) and features affected by a change might prove useful for automated synchronization. If automated patching with lightweight domain knowledge is applied from the start of a project, variants can be synchronized with high accuracy (93%) and almost perfect precision (97%).

In conclusion, existing patch techniques achieved good results on simulated variants of BusyBox, and have the potential to automate the synchronization of variants in clone-and-own. A potential that is further increased by lightweight domain knowledge. In the future, we will investigate to which extent this potential can be utilized in other projects.

References

[1] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. New York, NY, USA: ACM/Addison-Wesley, 2000.

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

[3] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines*. Berlin, Heidelberg, Germany: Springer, 2013. [Online]. Available: https://doi.org/10.1007/978-3-642-37521-7

[4] J. Rubin, K. Czarnecki, and M. Chechik, "Managing Cloned Variants: A Framework and Experience," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. New York, NY, USA: ACM, 2013, pp. 101–110.

[5] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An Exploratory Study of Cloning in Industrial Software Product Lines," in *Proc. Europ. Conf. on Software Maintenance and Reengineering (CSMR)*. Washington, DC, USA: IEEE, 2013, pp. 25–34.

[6] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, S. Stănciulescu, A. Wąsowski, and I. Schaefer, "Flexible Product Line Engineering with a Virtual Platform," in *Proc. Int'l Conf. on Software Engineering (ICSE)*. New York, NY, USA: ACM, 2014, pp. 532–535.

[7] S. Stănciulescu, S. Schulze, and A. Wąsowski, "Forked and Integrated Variants in an Open-Source Firmware Project," in *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME)*. IEEE, Sep. 2015, pp. 151–160.

[8] R. Lapeña, M. Ballarin, and C. Cetina, "Towards Clone-and-Own Support: Locating Relevant Methods in Legacy Products," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. New York, NY, USA: ACM, 2016, pp. 194–203.

[9] L. Linsbauer, S. Fischer, R. E. Lopez-Herrejon, and A. Egyed, "Using Traceability for Incremental Construction and Evolution of Software Product Portfolios," in *Proc. Int'l Symposium on Software and Systems Traceability (SST)*. Piscataway, NJ, USA: IEEE, 2015, pp. 57–60.

[10] T. Kehrer, "Calculation and Propagation of Model Changes Based on User-Level Edit Operations: A Foundation for Version and Variant Management in Model-Driven Engineering," Ph.D. dissertation, University of Siegen, Germany, 2015.

[11] W. Mahmood, D. Strueber, T. Berger, R. Laemmel, and M. Mukelabai, "Seamless Variability Management With the Virtual Platform," in *Proc. Int'l Conf. on Software Engineering (ICSE)*. Piscataway, NJ, USA: IEEE, 2021, pp. 1658–1670.

[12] G. K. Michelon, "Evolving System Families in Space and Time," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. New York, NY, USA: ACM, 2020, pp. 104—-111. [Online]. Available: https://doi.org/10.1145/3382026.3431252

[13] J. Krüger and T. Berger, "An empirical analysis of the costs of clone- and platform-oriented software reuse," in *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. New York, NY, USA: ACM, 2020, pp. 432–444.

[14] S. Zhou, Ş. Stănciulescu, O. Leßenich, Y. Xiong, A. Wąsowski, and C. Kästner, "Identifying Features in Forks," in *Proc. Int'l Conf. on Software Engineering (ICSE)*. New York, NY, USA: ACM, May 2018, pp. 105–116. [Online]. Available: https://dl.acm.org/citation.cfm?id=3180205

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

[16] T. Kehrer, T. Thüm, A. Schultheiß, and P. M. Bittner, "Bridging the Gap Between Clone-and-Own and Software Product Lines," in *Proc. Int'l Conf. on Software Engineering (ICSE)*. Piscataway, NJ, USA: IEEE, May 2021, pp. 21–25.

[17] P. M. Bittner, A. Schultheiß, T. Thüm, T. Kehrer, J. M. Young, and L. Linsbauer, "Feature Trace Recording," in *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. New York, NY, USA: ACM, Aug. 2021, pp. 1007–1020. [Online]. Available: https://doi.org/10.1145/3468264.3468531

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

[19] T. Schmorleiz and R. Lämmel, "Similarity Management via History Annotation," in *Proc. Seminar on Advanced Techniques and Tools for Software Evolution (SATToSE)*. Dipartimento di Informatica Università degli Studi dell'Aquila, L'Aquila, Italy, Jul. 2014, pp. 45–48.

[20] T. Pfofe, T. Thüm, S. Schulze, W. Fenske, and I. Schaefer, "Synchronizing Software Variants with VariantSync," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. New York, NY, USA: ACM, Sep. 2016, pp. 329–332.

[21] W. Ji, T. Berger, M. Antkiewicz, and K. Czarnecki, "Maintaining Feature Traceability with Embedded Annotations," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. New York, NY, USA: ACM, 2015, pp. 61–70.

[22] H. Abukwaik, A. Burger, B. K. Andam, and T. Berger, "Semi-Automated Feature Traceability with Embedded Annotations," in *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME)*. Piscataway, NJ, USA: IEEE, Nov. 2018, pp. 529–533.

[23] C. Sundermann, T. Thüm, and I. Schaefer, "Evaluating #SAT Solvers on Industrial Feature Models," in *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. New York, NY, USA: ACM, Feb. 2020.

[24] D. Beyer, "Competition on software verification," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2012, pp. 504–524.

[25] C. Kästner, K. Ostermann, and S. Erdweg, "A Variability-Aware Module System," in *Proc. Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. New York, NY, USA: ACM, Oct. 2012, pp. 773–792.

[26] N. Wells, "Busybox: A swiss army knife for linux," *Linux Journal*, vol. 2000, no. 78es, pp. 10–es, 2000.

[27] A. V. Rhein, J. Liebig, A. Janker, C. Kästner, and S. Apel, "Variability-aware static analysis at scale: An empirical study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 27, no. 4, pp. 1–33, 2018.

[28] A. Mordahl, J. Oh, U. Koc, S. Wei, and P. Gazzillo, "An empirical study of real-world variability bugs detected by variability-oblivious tools," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 50–61.

[29] R. Shahin and M. Chechik, "Automatic and efficient variability-aware lifting of functional programs," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–27, 2020.

[30] D. Reuling, U. Kelter, J. Bürdek, and M. Lochau, "Automated N-Way Program Merging for Facilitating Family-Based Analyses of Variant-Rich Software," *Trans. on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 3, pp. 13:1–13:59, Jul. 2019.

[31] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "The ECCO Tool: Extraction and Composition for Clone-and-Own," in *Proc. Int'l Conf. on Software Engineering (ICSE)*. Piscataway, NJ, USA: IEEE, 2015, pp. 665–668.

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

[33] D. Batory, "Feature Models, Grammars, and Propositional Formulas," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Berlin, Heidelberg, Germany: Springer, 2005, pp. 7–20.

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

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

[36] R. Conradi and B. Westfechtel, "Version Models for Software Configuration Management," *ACM Computing Surveys (CSUR)*, vol. 30, no. 2, pp. 232–282, Jun. 1998.

[37] T. Kehrer, U. Kelter, and G. Taentzer, "Consistency-Preserving Edit Scripts in Model Versioning," in *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. New York, NY, USA: ACM, 2013, pp. 191–201.

[38] T. Kehrer, U. Kelter, P. Pietsch, and M. Schmidt, "Adaptability of Model Comparison Tools," in *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. New York, NY, USA: ACM, 2012, pp. 306–309. [Online]. Available: https://doi.org/10.1145/2351676.2351731

[39] D. MacKenzie, P. Eggert, and R. Stallman, *Comparing and Merging Files with GNU diff and patch*. Network Theory Ltd., 2003.

[40] T. Mens, "A State-of-the-Art Survey on Software Merging," *IEEE Trans. on Software Engineering (TSE)*, vol. 28, no. 5, pp. 449–462, May 2002.

[41] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Bottom-Up Adoption of Software Product Lines: A Generic and Extensible Approach," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. New York, NY, USA: ACM, 2015, pp. 101–110.

[42] T. Ziadi, C. Henard, M. Papadakis, M. Ziane, and Y. Le Traon, "Towards a Language-Independent Approach for Reverse-Engineering of Software Product Lines," in *Proc. ACM Symposium on Applied Computing (SAC)*. New York, NY, USA: ACM, 2014, pp. 1064–1071.

[43] C. Kästner, K. Ostermann, and S. Erdweg, "A variability-aware module system," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, 2012, pp. 773–792.

[44] D. Reuling, U. Kelter, J. Bürdek, and M. Lochau, "Automated n-way program merging for facilitating family-based analyses of variant-rich software," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 3, pp. 1–59, 2019.

[45] A. Schultheiß, P. M. Bittner, S. El-Sharkawy, T. Thüm, and T. Kehrer, "Simulating the Evolution of Clone-and-Own Projects with VEVOS," in *Proc. Int'l Conf. on Evaluation Assessment in Software Engineering (EASE)*. New York, NY, USA: ACM, 2022, p. 231–236. [Online]. Available: https://doi.org/10.1145/3530019.3534084

[46] K. Schmid, S. El-Sharkawy, and C. Kröher, "Improving software engineering research through experimentation workbenches," in *From Software Engineering to Formal Methods and Tools, and Back*. Springer, 2019, pp. 67–82.

[47] C. Kästner, "Differential testing for variational analyses: Experience from developing kconfigreader," *arXiv preprint arXiv:1706.09357*, 2017.

[48] T. Berger, S. She, R. Lotufo, K. Czarnecki, and A. Wasowski, "Feature-to-code mapping in two large product lines." in *SPLC*. Citeseer, 2010, pp. 498–499.

[49] J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, T. Leich, and G. Saake, *Mastering Software Variability with FeatureIDE*. Berlin, Heidelberg, Germany: Springer, 2017.

[50] D. Le Berre and A. Parrain, "The Sat4j Library, Release 2.2," *J. Satisfiability, Boolean Modeling and Computation*, vol. 7, no. 2-3, pp. 59–64, 2010.

[51] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. New York, NY, USA: ACM, 1999, vol. 463.

[52] A. Schultheiß, P. M. Bittner, T. Thüm, and T. Kehrer, "Artifact for Quantifying the Potential to Automate the Synchronization of Variants in Clone-and-Own," Jul. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6790199

[53] B. Klatt, M. Küster, and K. Krogmann, "A Graph-Based Analysis Concept to Derive a Variation Point Design from Product Copies," in *Proc. Int'l Workshop on Reverse Variability Engineering (REVE)*, Mar. 2013, pp. 1–8.

[54] A. Schultheiß, P. M. Bittner, T. Kehrer, and T. Thüm, "On the Use of Product-Line Variants as Experimental Subjects for Clone-and-Own Research: A Case Study," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. New York, NY, USA: ACM, 2020.

[55] A. E. Hassan and R. C. Holt, "Predicting change propagation in software systems," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 2004, pp. 284–293.

[56] H. Malik and A. E. Hassan, "Supporting software evolution using adaptive change propagation heuristics," in *2008 IEEE International Conference on Software Maintenance*. IEEE, 2008, pp. 177–186.

[57] M. Chechik, W. Lai, S. Nejati, J. Cabot, Z. Diskin, S. Easterbrook, M. Sabetzadeh, and R. Salay, "Relationship-based change propagation: A case study," in *2009 ICSE Workshop on Modeling in Software Engineering*. IEEE, 2009, pp. 7–12.

[58] H. K. Dam and M. Winikoff, "Supporting change propagation in uml models," in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–10.

[59] H. Kagdi, M. Gethers, and D. Poshyvanyk, "Integrating Conceptual and Logical Couplings for Change Impact Analysis in Software," *Empirical Software Engineering (EMSE)*, vol. 18, no. 5, pp. 933–969, 2013.

[60] N. Lodewijks, "Analysis of a clone-and-own industrial automation system: An exploratory study," *SATToSE*, 2017.

[61] M. Staples and D. Hill, "Experiences Adopting Software Product Line Development without a Product Line Architecture," in *Proc. Asia-Pacific Software Engineering Conference (APSEC)*. Washington, DC, USA: IEEE, 2004, pp. 176–183.

[62] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach," *Science of Computer Programming (SCP)*, vol. 74, no. 7, pp. 470–495, May 2009.

[63] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Clone Management for Evolving Software," *IEEE Trans. on Software Engineering (TSE)*, vol. 38, no. 5, pp. 1008–1026, Sep. 2012.

[64] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, 2005, pp. 187–196.

[65] L. Aversano, L. Cerulo, and M. Di Penta, "How clones are maintained: An empirical study," in *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*. IEEE, 2007, pp. 81–90.

[66] C. J. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, pp. 645–692, 2008.

[67] N. Göde and R. Koschke, "Frequency and risks of changes to clones," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 311–320.

[68] D. Wille, T. Runge, C. Seidl, and S. Schulze, "Extractive Software Product Line Engineering Using Model-based Delta Module Generation," in *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. New York, NY, USA: ACM, 2017, pp. 36–43.

[69] D. Wille, M. Tiede, S. Schulze, C. Seidl, and I. Schaefer, "Identifying Variability in Object-Oriented Code Using Model-Based Code Mining," in *Proc. Int'l Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, T. Margaria and B. Steffen, Eds., 2016, pp. 547–562.

[70] A. Schlie, D. Wille, S. Schulze, L. Cleophas, and I. Schaefer, "Detecting Variability in MATLAB/Simulink Models: An Industry-Inspired Technique and Its Evaluation," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. New York, NY, USA: ACM, 2017, pp. 215–224.

[71] J. Martinez, T. Ziadi, M. Papadakis, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Feature Location Benchmark for Software Families Using Eclipse Community Releases," in *Proc. Int'l Conf. on Software Reuse (ICSR)*. Berlin, Heidelberg, Germany: Springer, Jun. 2016, pp. 267–283.

[72] W. Fenske, J. Meinicke, S. Schulze, S. Schulze, and G. Saake, "Variant-Preserving Refactorings for Migrating Cloned Products to a Product Line," in *Proc. Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER)*. Piscataway, NJ, USA: IEEE, 2017, pp. 316–326.

[73] J. H. Weber, A. Katahoire, and M. Price, "Uncovering Variability Models for Software Ecosystems from Multi-Repository Structures," in *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. New York, NY, USA: ACM, 2015, pp. 103:103–103:108.