Synchronizing Software Variants with VariantSync

Tristan Pfofe  
ckc group  
Germany

Thomas Thüm  
TU Braunschweig  
Germany

Sandro Schulze  
TU Hamburg-Harburg  
Germany

Wolfram Fenske  
University of Magdeburg  
Germany

Ina Schaefer  
TU Braunschweig  
Germany

ABSTRACT
Developing and managing software variants is a key challenge in today’s software development. Due to conflicting requirements, software is developed in multiple variants to satisfy the needs of individual customers. While software product lines allow the efficient development of a high number of variants, many projects in industrial software development start with few variants, where each variant is developed separately. Unfortunately, for an increasing number of variants, this clone-and-own approach becomes error-prone and unprofitable regarding synchronization of changes between variants. With VariantSync, we demonstrate a tool to reduce the gap between clone-and-own and product lines by automating the synchronization of software variants and simplifying a potential later transition to a product line.

CCS Concepts
•Software and its engineering → Software configuration management and version control systems; Software maintenance tools;

Keywords
Clone-and-own; evolution, software product lines

1. INTRODUCTION
To overcome the increasing demand for tailored software systems, industrial software development often uses clone-and-own to build a new variant by copying and adapting an existing variant. Indeed, this procedure is easy to use and requires less up-front investments. However, with an increasing number of variants, development becomes redundant and the maintenance effort rapidly grows [8, 6, 33]. In practice, clone-and-own approaches can be realized with well-known development tools, e.g., with version control systems by creating a new branch for each variant based on another branch [7, 29, 27, 4, 28, 32, 30]. For an increasing number of branches, tracking all changes and merging them into the appropriate variants becomes tedious and error-prone. Hence, at some point, a sufficient number of variants is reached and the migration to a product line is necessary. In this process, all differences that occurred since the creation of the first variant need to be recovered. To ease this migration, some approaches provide partial automation [3, 2, 14, 25, 26, 13, 18]. Nevertheless, the effort for manual reengineering remains high, the migration needs to interrupt the development process [11], and currently used tools need to be replaced with product-line tools [10].

However, using a product line to develop variants has several downsides. First, product lines have high up-front investments which make the development of few variants unprofitable [19, 24]. Hence, introducing a product line would be a risky task that could not pay off if the number of required variants is unknown at beginning of development. Second, tool support for product lines is not as mature as it is for single-system. Most tools are based on techniques that use a common code base to generate variants by specifying a feature selection [5]. In practice, one of the most used techniques are preprocessors. They use directives to exclude code fragments. Tools for programming languages typically do not support any reasoning on them because they are oblivious to directives. Indeed, new tools for type checking [15] and refactoring [16] of preprocessors have been built. Nevertheless, our experience in building tool support for product lines [31, 20] is that, in principle, it is possible to provide good tool support for product lines, but in practice, those tools will always be a step behind compared to single-system tools. Thus, a small number of variants is typically developed with clone-and-own instead of product-line engineering [14, 13, 32, 18, 12, 9, 17].

With VariantSync, we developed an open-source Eclipse plug-in distributed under L-GPL\textsuperscript{1} to improve the development of few variants. Variants are developed separately as in clone-and-own and changes are automatically propagated using domain knowledge of developers. VariantSync logs changes during development, tags these changes to feature expressions, and automates the synchronization of changes between variants. As a side effect, using VariantSync for a while may increase the feature-to-code mapping and, thus, ease a potential later migration to a product line.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

\textsuperscript{1}https://github.com/tthuem/VariantSync
2. IDENTIFICATION OF SYNCHRONIZATION TARGETS

The automated identification of valid synchronization targets for a given change is a challenging task. We use domain knowledge in terms of feature expressions to detect appropriate variants that need to be synchronized with a change that occurred in another variant.

In Figure 1, we introduce feature model and feature configurations of three variants that serve as our running example. Each variant is a search engine which provides functions to search specific contents inside HTML, TXT, or LATEX files. Variants contain a class named `ContentHandler`, which indexes files and checks file types.

2.1 Feature Modeling

In order to identify valid synchronization targets, `VariantSync` maps each change to a feature expression. A feature expression is a propositional formula over a set of features [1]. Technically, besides those expressions, `VariantSync` only needs a set of features for each variant specified by the user. However, to avoid error-prone text files containing feature names, to profit from established variability modeling tools, and to ease variant development, we reuse the infrastructure of FeatureIDE [31]. That is, `VariantSync` uses a feature model that specifies all features and their relationships for all variants. In particular, we reuse the graphical, tree-based, and textual editor for feature modeling [31]. Based on that feature model, each variant is characterized by a set of features. For this purpose, we reuse the configuration editor of FeatureIDE to support developers in choosing a feature selection that is valid, as shown in Figure 1. Feature model and configurations are stored as an ordinary FeatureIDE project located in the same workspace as the variants.

Moreover, we adapt FeatureIDE’s editor for cross-tree constraints, which supports developers in creating propositional constraints on features by providing syntax checking and a content assist [31]. With our adaptation, developers can specify and store feature expressions.

2.2 Tagging Code to Features

`VariantSync` identifies relevant changes for distinct variants by tagging each changed code fragment to a feature expression. To avoid code pollution, we store the tagged information as metadata instead of using annotations, as done by preprocessor-based product-line tools [5, 12]. Tagged code fragments are visualized with a distinct background color, as shown in Figure 2.

3. AUTOMATED SYNCHRONIZATION

After detecting changes and tagging them to feature expressions, `VariantSync` computes valid synchronization targets and provides synchronization from different views.

3.1 Computing Synchronization Targets

For a given change with its feature expression, compatible synchronization targets are those variants for which the feature expression evaluates to true. We evaluate feature expressions for a given configuration using FeatureIDE [31]. Based on the synchronization compatibility, we classify variants into three classes of synchronization targets [23]. The first class are variants without merge conflicts. A change can be lexically merged into these variants without a merge conflict, but a synchronization could be wrong from a requirements perspective because it is undefined whether these variants need the changes. `Variants to synchronize` form the second class. These variants implement the feature expression, but a merge could cause a conflict which can only be solved manually. The third class are variants for automatic synchronization. These variants are targets and allow an automated synchronization without merge conflicts. For example, feature `TXT` of Variant 2 was changed by adding the new method `indexTXTFile`. This change can be lexically merged into Variant 3 and the synchronization is correct from a requirements perspective because Variant 3 also implements feature `TXT`.

3.2 Decoupled Synchronization

In order to perform the synchronization, we separate the process of implementation from synchronization. In this
way, it is possible that one developer works on a variant and an expert of the synchronization target performs the synchronization. This decoupled synchronization allows to synchronize variants from different points of view. For this purpose, VariantSync provides two Eclipse views: the target-focused and source-focused synchronization view (Figure 3). The target-focused synchronization focuses on updating single variants. Using the target-focused view, a developer chooses the variant that he wants to update. Then, the developer can choose a feature expression and the view determines whether changes can be automatically merged into the chosen variant and the developer can start the automatic or manual synchronization. To summarize, this process is similar to the development of branches in version control systems, e.g., using pull requests to synchronize branches. As a difference and related to cherry picking mechanisms [22], we only merge distinct changes instead of commits. The source-focused synchronization focuses on propagating changes into a set of target variants. Working with this view (cf. Figure 3), the developer chooses a feature expression and the view determines all code changes that are tagged to the chosen feature expression, ordered by files and timestamps. After selecting changes, the view provides all valid target variants and checks whether changes can be automatically merged into target variants or not. As an application example, the source-focused synchronization adjusts release cycles of variants because each variant receives changes immediately after the change occurred. If a variant should be released, then this variant already contains all relevant changes that occurred in the meantime in other variants. Synchronizing a bundle of changes with one of the views requires considerable interaction between developer and view. To reduce this effort, a batch mode automates the process of change propagation. As shown in Figure 4, the developer only needs to select a set of feature expressions (source-focused) or a set of variants (target-focused) to start the synchronization. Then, the batch mode propagates all changes into valid synchronization targets or merges all changes into selected variants in the order, in which the changes occurred originally. Nevertheless, in case of merge conflicts, the developer still has to solve the conflicts manually.

During synchronization, merging is necessary to integrate changes of one variant into another variant. We define a change as the addition or removal of files and folders, or as the modification of a file where consecutive changed lines are counted as one change. VariantSync uses a lexical merge together with the three-way merge technique [21]. To detect merge conflicts and to perform merges, we use java-diff-utils.2 If merge conflicts occur, VariantSync opens the Eclipse Compare Editor to provide a user interface for manual conflict resolution.

3.3 Increasing Feature-to-Code Mapping

To increase the feature-to-code mapping, VariantSync automatically tags code that was merged into a target variant with the feature expression of the change. Before the merge process starts, VariantSync activates the feature-expression context of the current change. Then, VariantSync merges the changes into the target variants. The feature-expression context notices these code changes in the target variants and tags the code to the feature expression. That is, VariantSync synchronizes changes and tagging. The advantage is that the feature expression for code is also available in target variants and the overall feature-to-code mapping is significantly increased.

4. CONCLUSION

With VariantSync, we propose an Eclipse plug-in to enhance variant development with clone-and-own and to ease the migration to a product line. On the one hand, VariantSync makes variant development more efficient by automating the identification of synchronization targets with tagging in feature-expression contexts. Furthermore, VariantSync decouples implementation from synchronization by providing a synchronization from a target-focused or source-focused point of view. On the other hand, VariantSync eases the transition to a product line by establishing a feature-to-code mapping over time.

5. ACKNOWLEDGMENTS

Many thanks to Lei Luo for his help with the development of an initial version of VariantSync.

6. REFERENCES


2https://github.com/KengoTODA/java-diff-utils


