Using Traceability for Incremental Construction and Evolution of Software Product Portfolios

Lukas Linsbauer, Stefan Fischer, Roberto E. Lopez-Herrejon and Alexander Egyed
Johannes Kepler University
Linz, Austria
{lukas.linsbauer, stefan.fischer, roberto.lopez, alexander.egyed}@jku.at

***Abstract*—Software reuse has become mandatory for companies to compete and a wide range of reuse techniques are available today. Despite the great benefits of these techniques, they also have the disadvantage that they do not necessarily support the creation and evolution of closely related products – products that are customized to different infrastructures, ecosystems, machinery, or customers. In this paper we outline an approach for incrementally constructing and evolving software product portfolios of similar product variants. An initial proof of concept demonstrates its feasibility.**

I. Introduction

Many companies do not build one-of-a-kind software products. Rather they build a portfolio of similar products, often tailored to different customer needs, machinery, ecosystems, or other infrastructures. The number of product variants is variable but in our experience we have observed anything between a handful and 1000+ variants and correspondingly large code sizes. These product variants share a high degree of common functionality (i.e. features) but still differ. Software Product Lines (SPLs) are meant to address this problem by providing a single, configurable system from which all desired product variants can be derived. Unfortunately, SPLs require considerable upfront investments [1]. Moreover, due to continuing technological changes it is impossible to predict all the product variants required in the future nor is it possible to avoid frequent evolutionary changes – both still open challenges for current SPL approaches [2].

In practice we rarely observed full-blown SPLs. Instead, we found that companies often resorted to an ad-hoc practices generally referred to as clone-and-own. In clone-and-own, a new product variant is created by modifying existing variants that closely match the new variant's needs while copy and pasting from other variants as needed. Clone-and-own has three informal steps: 1) locating reusable artifacts (e.g. code) in the existing variants and 2) copying/merging those artifacts that closest match the requirements into a new product variant, 3) adapting the new product variant to account for requirements that did not exist thus far in any existing variant.

While building new product variants is a considerable challenge, maintaining the existing ones is even more so. Companies that practice clone-and-own usually do not keep track of the copying of code, which makes it harder to take advantage of reuse opportunities. Furthermore, the modification of a feature (e.g. bug fix) needs to be applied to all variants which share that feature. Each modified variant then has to be re-tested and possibly re-deployed manually at some remote customer site, infrastructure, or machinery.

In this paper we propose an approach for constructing, maintaining, and evolving a software products portfolio consisting of an arbitrary number of variants. The novelty of our approach is that it provides a bridge between the ad-hoc clone-and-own and a meticulously planned SPL, by leveraging their respective advantages and mitigating their drawbacks. At its core, our approach relies on traceability information that maps features and feature interactions (the building blocks of the system variants) to the artifacts that implement them. No upfront investment is necessary, because companies almost always have an existing portfolio of variants and our approach supports a one-time automated initialization of pre-existing variants. Once initialized, the approach is incremental where an engineer may create a new variant, based on automated reuse from existing variants that our approach provides. Manual finalization of a product variant is guided by hints that our approach generates if the automated reuse is not capable of completing the desired product. A finalized product may then be added to our approach such that its manual modifications become reusable downstream. The more products are added, the more complete and correct the approach becomes. The advantage of our approach is that developers do not have to change their development practices. Software engineers can continue to develop single product variants the way they are used to but get automated support in doing so. However, they can also modify the entire portfolio of variants if they like (e.g. bug fix across multiple variants) instead of having to do so separately for each variant. Our approach assumes that product variants exhibit similar code structures (i.e. a common architecture) which is a valid assumption based on our experiences thus far (SPLs also make this assumption).

We performed an evaluation on parts of our approach in our previous work [3]. This paper gives a broader vision of our intended work flow and differs from and extends our previous work in the evolution of features and product variants.

II. Approach Architecture

This section presents our incremental approach for creating, maintaining, and evolving product portfolios. Figure 1 provides a high level overview of its four major workflow cycles. The first cycle *initializes* the system based on existing product variants (if available), it extracts traces between features and feature interactions and their implementing artifacts across the existing variants. The second cycle *constructs* product variants based on previously extracted information. Since composing new product variants that did not exist beforehand may result in partially incomplete composed variants, human effort may be required to finalize them. Here our approach automatically composes the parts that do exist and provides hints for the parts that need adaptation. Once a human has finalized a product variant, the third cycle allows the finalized variant to be extracted back in order to *extend* the product portfolio for future use. The extension is not just an incremental form of the initialization but is complicated by the existence of evolutionary changes between variants. Finally, the fourth cycle allows a human *modifying* the extracted product portfolio directly (e.g. applying a bug fix to a feature) to then automatically recompose all the affected product variants (rather than fixing all products separately). The modification may have two forms: 1) modifying the knowledge base directly or 2) modifying a single variant and then updating/replacing the older variant in the knowledge base. The *Initialization* is an optional first step in case of pre-existing product variants. *Construction*, *Extension*, and *Modification* are then arbitrarily repeatable steps to incrementally evolve the product portfolio.

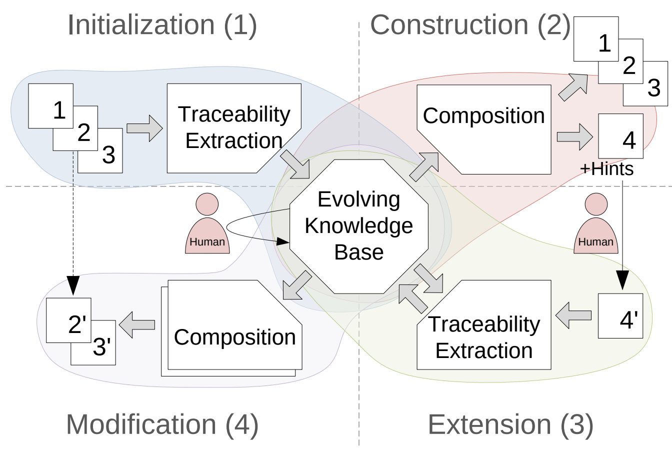

Fig. 1: Cycles Overview

At the core of our approach is an evolving knowledge base of reusable artifacts and traces to the features or feature interactions they implement. The two major components then populate and use this knowledge base: i) *Traceability Extraction* automatically extracts traces among the features and artifacts (e.g. between features, source code, configurations, test cases, or design) complete with dependencies, variability models and other useful information, and ii) *Composition* automatically composes new product variants by selecting from their features. We argue that the extraction and composition can handle any kind of artifact - from source code, configurations, test cases and even documentation - though the proof of concept and technical paper demonstrated this for source code only [3]. Next, we describe the four cycles and their two basic building blocks by means of an abstract example shown in Figure 2.

A. Initialization

This first step *initializes* the knowledge base with existing product variants.

Input: Variants. As input, it takes an arbitrary number of existing product variants. A product variant should consist of a list of features it implements (this list is arbitrarily definable) and its implementation artifacts (e.g. source code, test code, documentation, or configuration). Note that determining the features for product variants is not in the scope of this work. We assume that they are already known. As an example, consider a drawing application portfolio which supports three product variants ① in Figure 2. The three initial product variants have features for drawing shapes and also a feature for coloring them. The first variant shown at the top left corner contains features `LINE` ╱, `CIRCLE` ○ and `RECTANGLE` □. The second variant in the middle has features `LINE` ╱, `TRIANGLE` △ and `COLOR` and the third variant has features `LINE` ╱, `CIRCLE` ○ and `TRIANGLE` △. The product variants also have implementation artifacts (not shown in the figure).

Process: Traceability Extraction. The extraction ② compares the features and artifacts of any two input product variants to determine their commonalities and differences. For example, one rule says that if two products have features in common then it is presumed that the artifacts they have in common trace to these features (i.e. these artifacts implement these features). Moreover, the extraction has rules to recognize feature interactions [4]. For example, there is an interaction in the implementation between feature `COLOR` and feature `LINE` and `TRIANGLE` because some portion of their implementation changes when their lines have to be colored. Furthermore, it extracts dependencies among the features (e.g. feature `RECTANGLE` requires feature `LINE` to draw its sides). Finally, it extracts dependencies among the artifacts (e.g. a method requires the class who owns it) and their ordering (e.g. the statement order in a method matters). All this information is stored in the knowledge base. The extraction works best if the variants implement the features consistently. If, for example, two variants share a feature but this feature was implemented by two different development teams then the implementations would likely differ widely and our approach would fail. Our approach presumes that features are only ever implemented once and then reused across variants. In this case, our approach would recognize implementation cloning as shared features.

Result: Knowledge Base. The result of the extraction is the knowledge base ③.

B. Construction

The construction uses the knowledge base to either recompose existing product variants or to compose new product variants with new features or feature combinations.

Process: Composition. The composition ④ ⑤ is to a greater extent the reverse process of the extraction. It merges extracted fragments together based on selected features. It

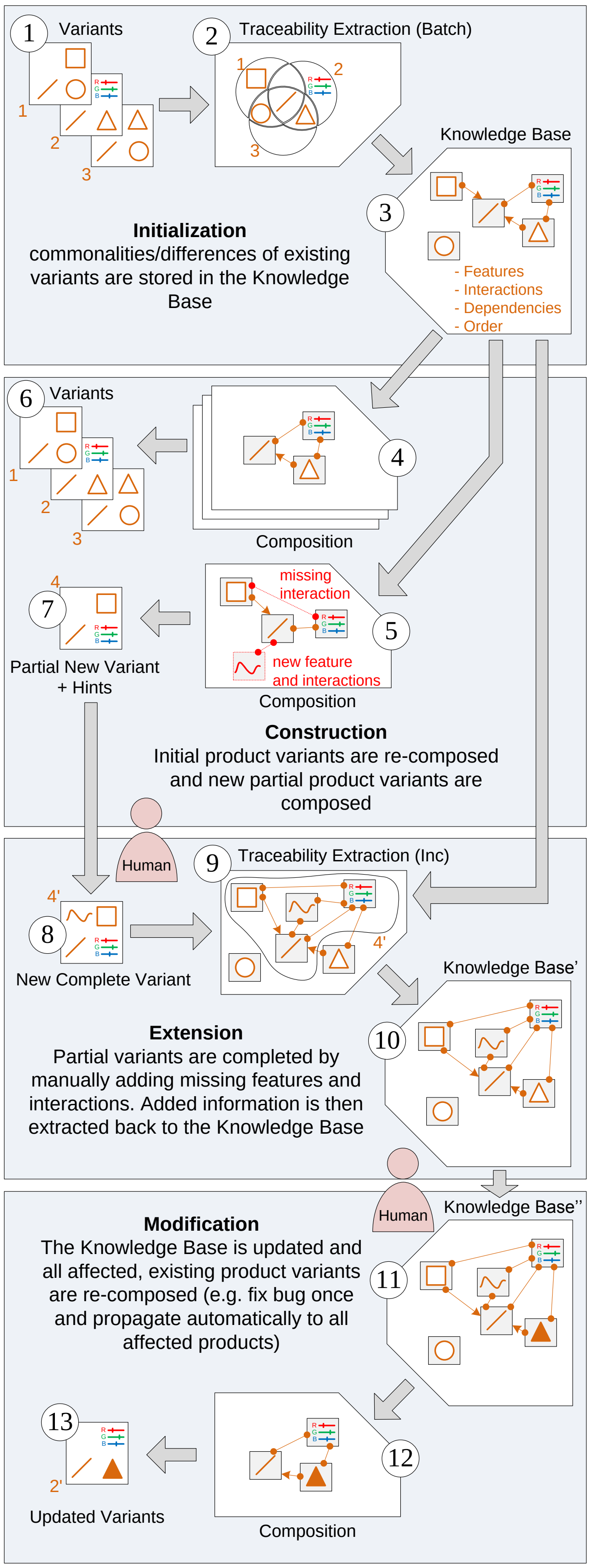
1 Variants
2 Traceability Extraction (Batch)
Knowledge Base
3
- Features
- Interactions
- Dependencies
- Order
Initialization
commonalities/differences of existing variants are stored in the Knowledge Base
6 Variants
4
Composition
missing interaction
5
new feature and interactions
Composition
7
Partial New Variant + Hints
Construction
Initial product variants are re-composed and new partial product variants are composed
Human
9 Traceability Extraction (Inc)
4'
8
New Complete Variant
Knowledge Base'
10
Extension
Partial variants are completed by manually adding missing features and interactions. Added information is then extracted back to the Knowledge Base
Human
Knowledge Base''
Modification
The Knowledge Base is updated and all affected, existing product variants are re-composed (e.g. fix bug once and propagate automatically to all affected products)
11
13
2'
Updated Variants
12
Composition

Fig. 2: Example Sequence

takes into account not only the single features that were selected but also their interactions, dependencies, and ordering.

Result 1: Existing Variant. When selecting sets of features for the composition of product variants ⑥ that already existed in the input products then all the necessary information to compose these variants will be available in the knowledge base. In such a case, the product variants are composable fully automatically. This is useful when the initial product variants are not stored separately (as in classical SPLs) or are evolved later (see Modification in II-D).

Result 2: New Variant. The most important application of our approach is the ability to compose new product variants in part/full out of features from previously existing product variants. For example, consider composing a new product ⑦ with the previously known features `LINE`, `RECTANGLE`, `COLOR` and the new feature `CURVE` ∿. Our approach first automatically composes the three known features by taking them from the knowledge base. Even though these features existed before, `RECTANGLE` and `COLOR` never appeared together in a single product variant. Hence the feature implementations may be known but not necessarily all relevant feature interactions (e.g. is the coloring of a rectangle different from the coloring of a line?). The composition thus computes a list of potentially missing feature interaction. Also for new features, such as `CURVE`, there is no knowledge to be exploited. For such missing features, the composition also provides hints. A human is then expected to finalize these partial product variants. These partial products get better the more product variants already exist.

C. Extension

Our approach expands its knowledge base with every new product variant. A manually finalized product variant ⑧ can and should be extracted back into the knowledge base, thus effectively evolving the knowledge base, to enable future compositions to reuse new features and interactions.

Input: New Finalized Variant. Based on the hints provided by the automated composition process (e.g. about missing features and feature interactions or violated dependencies between fragments) the software engineer can complete the new product variant ⑧.

Process: Traceability Extraction. Here, the extraction ⑨ follows an incremental model that uses the traceability information in the knowledge base as input and compares it with the new product variant to update and refine the stored traces. By using the new completed variant ⑧ as input for the extraction, we obtain knowledge about the feature `CURVE` and the interaction between features `RECTANGLE` and `COLOR`. As was mentioned earlier, the extension is not just an incremental form of the initialization but may be complicated by evolutionary changes between variants. Imagine the newly composed variant above also improved how lines are drawn. There are thus two versions of the feature `LINE` and without additional guidance, the extraction might confuse `LINE` with the new feature `CURVE`. In practice, we find companies do understand changes in product variants - especially if they are documented immediately and extracted as proposed on our approach.

Result: Knowledge Base. The result of this extraction is an evolved knowledge base $\textcircled{10}$ which now contains, in addition to the previously extracted information, the `CURVE` feature and additional feature interactions. It may even be desired to maintain different versions of a feature. For example, an existing customer may not be issued a more efficient version of `LINE` without paying for it but perhaps an existing customer would be issued bug fixes to features.

D. Modification

Lastly a software engineer can manually evolve the knowledge base (e.g. to apply a bug fix to a feature/artifact) and then propagate the changes to all affected variants by automatically re-composing them. This avoids having to apply the same change to all the affected variants separately.

These manual modifications can either be performed directly on the knowledge base or by means of a special product variant to which the changes are applied and which is then fed back through the extraction to update the knowledge base.

Input: Knowledge Base. For example, consider a change to feature `TRIANGLE` in combination with feature `COLOR` (a feature interaction) that allows not only to color the edges, but also to fill the triangle with a color. A software engineer can identify the features and interactions responsible for this functionality using the knowledge base and manually adapt the implementation in the knowledge base $\textcircled{11}$.

Process: Composition. Subsequently all the affected product variants $\textcircled{13}$ can be automatically determined and re-composed $\textcircled{12}$ using the updated knowledge base $\textcircled{11}$.

Result: Updated Variants. Here only product variant 2 had to be updated $\textcircled{13}$. Had there been more variants implementing the affected features and feature interaction then all those variants would be re-computable instantly.

III. Proof of Concept & Evaluation

To demonstrate the basic feasibility of our approach, we built a proof-of-concept tool that was successfully applied on five case studies ranging from 12 to 256 variants with up to 344 KLOC in code size. In our evaluation we used a random subset of the available product variants for each case study as input to the extraction (a subset of the 12 to 256 variants) and subsequently composed all its remaining variants, and computed precision and recall to determine artifacts surplus or missing respectively. We found that less than 20% of the existing product variants as input allowed for the near optimal construction of new product variants (the other 80% of available products, that were reconstructed by reusing existing functionality). But even before these 20% of possible product variants are reached the approach is useful in providing reusable artifacts for new product variants. Our proof-of-concept is currently limited to Java only, and focused on two of the four cycles discussed here. The details of this evaluation and the two cycles are published in [3].

IV. Related Work

Rubin et al. proposed a *conceptual* framework for managing product variants that are the result of clone-and-own practices [5]. They outlined a series of operators that described the set of processes and activities to manage software variants in the different scenarios they encountered in their case studies. The extraction and composition processes presented in this work could be seen as an *actual implementation* of some of their proposed operators. However, our work proposes a work flow for how our operations could be used for different purposes, like evolution and maintenance.

There are several other traceability and information mining algorithms as for example presented by Capobianco et al. in [6] or by Kagdi et al. in [7]. Ziadi et al. propose an approach for identifying features through differences in product variants [8]. The difference to our work is that Ziadi et al. work on a model as product abstraction and do not map features directly to source code as proposed in this work, which also leads to a different level of granularity of their mapping.

V. Conclusions

We presented an approach for constructing and evolving software product portfolios. It provides automated reuse of existing arbitrary development artifacts and supports software engineers during the maintenance and evolution of product portfolios. The approach extract traces from system variants that map features and feature interactions to their implementation artifacts. Based on these traces four usage cycles are proposed. A proof of concept evaluating five case studies for two of the four cycles demonstrated the feasibility of our approach in principle. We are currently applying this work on two industrial case studies and future work will focus on exploring more heterogeneous development artifacts.

Acknowledgments

This research was funded by the Austrian Science Fund (FWF) projects P25289-N15 and P25513-N15.

References

[1] G. Botterweck and A. Pleuss, “Evolution of software product lines,” in *Evolving Software Systems*, 2014, pp. 265–295.

[2] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, S. Stanciulescu, A. Wasowski, and I. Schaefer, “Flexible product line engineering with a virtual platform,” in *ICSE Companion*, 2014, pp. 532–535.

[3] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, “Enhancing clone-and-own with systematic reuse for developing software variants,” in *ICSME*, 2014, pp. 391–400.

[4] L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, “Recovering traceability between features and code in product variants,” in *SPLC*, 2013.

[5] J. Rubin, K. Czarnecki, and M. Chechik, “Managing cloned variants: a framework and experience,” in *SPLC*, 2013.

[6] G. Capobianco, A. D. Lucia, R. Oliveto, A. Panichella, and S. Panichella, “Improving ir-based traceability recovery via noun-based indexing of software artifacts,” *Journal of Software: Evolution and Process*, vol. 25, no. 7, pp. 743–762, 2013.

[7] H. H. Kagdi, M. Gethers, and D. Poshyvanyk, “Integrating conceptual and logical couplings for change impact analysis in software,” *Empirical Software Engineering*, vol. 18, no. 5, pp. 933–969, 2013.

[8] T. Ziadi, L. Frias, M. A. A. da Silva, and M. Ziane, “Feature identification from the source code of product variants,” in *CSMR*, T. Mens, A. Cleve, and R. Ferenc, Eds. IEEE, 2012, pp. 417–422.