A Framework for Managing Cloned Product Variants

Julia Rubin[§*] and Marsha Chechik[*]
[*]University of Toronto, Canada
[§]IBM Research at Haifa, Israel
mjulia@il.ibm.com, chechik@cs.toronto.edu

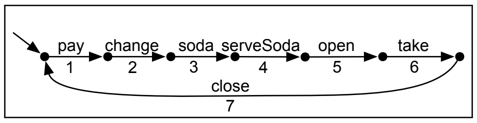
***Abstract*—We focus on the problem of managing a collection of related software products realized via cloning. We contribute a framework that explicates operators required for developing and maintaining such products, and demonstrate their usage on two concrete scenarios observed in industrial settings: sharing of features between cloned variants and re-engineering the variants into "single-copy" representations advocated by software product line engineering approaches. We discuss possible implementations of the operators, including synergies with existing work developed in seemingly unrelated contexts, with the goal of helping understand and structure existing work and identify opportunities for future research.**

I. Introduction

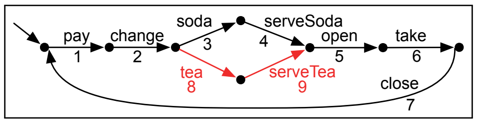
Software Product Line Engineering (SPLE) promotes strategic, well-managed software reuse [1]. In reality, however, products are developed ad-hoc, often using artifact cloning (the "clone-and-own" approach). Over the past decade, there have been individual attempts to propose solutions for dealing with cloned product variants. Some [2], [3] advocate refactoring such variants into software product line (SPL) representations (the product line *merge-refactoring*). Others [4], [5] propose efficient mechanisms for managing multiple variants without attempting to refactor them. The latter approaches are particularly important in light of our earlier exploratory study [6] which shows that organizations sometimes prefer to stick to cloning due to its simplicity, availability and developer independence. Despite numerous isolated solutions aiming to assist the practitioners in developing, maintaining and refactoring collections of related software products, this task is still challenging [6] and many gaps remain.

Inspired by an empirical analysis of realistic scenarios observed in industrial settings [7], this paper makes the following contributions. It (1) identifies the basic operators required for managing collections of cloned product variants (Sec. II); (2) demonstrates the applicability of the operators for addressing real-life scenarios (Sec. III); and (3) discusses possible implementations of the operators, including synergies with relevant solutions developed outside the SPLE context, and identifies remaining gaps (Sec. IV). The main goal of this paper is to structure existing work designed for or applicable to managing cloned product variants and to provide an agenda for future research.

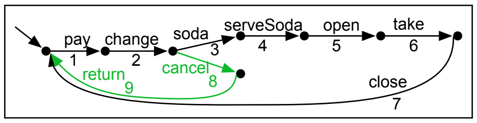
Example. We illustrate the presented operators and scenarios on a "toy" example of a set of related products realized via cloning, taken from [8]. Fig. 1 depicts four transition



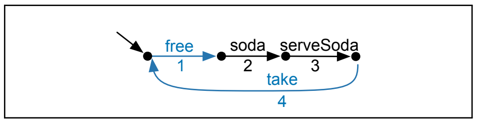
(a) Soda.



(b) Soda and Tea.



(c) Soda with Cancel.



(d) Free Soda.

Fig. 1. Vending Machine Variants.

systems corresponding to four beverage vending machine variants. The basic one, in Fig. 1(a), accepts payment, returns change, selects and serves soda. Then it opens a compartment allowing the user to take the drink, and, when taken, closes the compartment. A variant of this machine serving either soda or tea is shown in Fig. 1(b). It allows the user to choose the drink and then serves it. Yet another variant, in Fig. 1(c), allows to cancel the purchase before selecting the drink and returns the paid amount. The last, in Fig. 1(d), offers free drinks, and does not open or close the beverage compartment.

II. Notation and Operators

We define a *product* as a well-formed set of *artifacts*, such as code statements and model elements. A product's artifacts implement *features*. Inspired by Rajlich and Chen [9], we represent a product feature as a pair consisting of a *feature specification (FS)* – a label and a short description that identifies the feature, and a

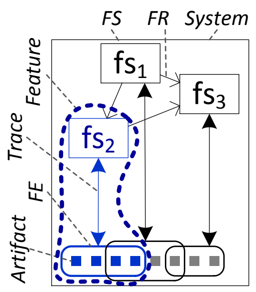


Fig. 2. Notations.

feature extension (FE) – a subset of product artifacts that realize the FS (see Fig. 2). A FS is *traced* to the FE that realizes it. Features can have *relationships (FR)* with each other, e.g., one feature can require another in order to operate. Finally, a *system* is a set of features (FSs traced to FEs) and relationships between them. A single system can correspond to an individual product or a set of related products (an SPL). In what follows, we use these notations to define a set of basic operators required for managing a collection of cloned products (see Table I).

`findFS` returns a set of feature specifications, i.e., <feature name, description> pairs, realized by the given product. For the vending machine in Fig. 1(b), the FSs include <soda, sells soda>, <tea, sells tea> and <pay, allows to pay for the drink being purchased>. To save space, in this paper we omit either the name or the description of a given FS, if clear from context.

`findFE`, commonly known as *feature location*, returns a FE of the given FS – the traced set of artifacts that realize the input FS. The exact form of the detected FE depends on the *goal* of feature location: e.g., "detect the artifacts that contribute only to the feature of interest" or "detect all artifacts required for the feature to be executable (including the main method of a program)". We declaratively represent this goal using the input *property* that specifies inclusion and exclusion conditions for the feature location process. For example, transitions 1 and 2 of the vending machine in Fig. 1(b) realize the pay FS w.r.t. the property which disregards transitions contributing to other features, such as soda and tea.

`require`? determines whether feature f_1 requires feature f_2 from the same product in order to operate. The input *property* captures the nature of the `require` dependency. Such a property can express simple dependencies such as "f_1 *requires* f_2 in order to compile", or more complex behavior dependencies. For our example in Fig. 1(b), the soda feature requires the pay feature w.r.t. the property "soda is served only after a payment is received". The operator returns a *set of witnesses*, each demonstrating the `require` relationship between the artifacts of f_1 and f_2 (or none if the features are independent). In the example above, a *witness* is the flow between the pay and the soda features: transitions 1 and 2, realizing the first one, precede transitions 3 and 4, realizing the second.

`agree`? determines whether feature f_1 of $product_1$ is consistent with feature f_2 of $product_2$, i.e., whether there are no *disagreements* in both the specifications and the extensions of the two seemingly equivalent features. For the four products in Fig. 1, the take drink feature, allowing one to take the ordered drink, is implemented similarly in Figs. 1(a), 1(b) and 1(c), by transitions 5-7. However, this feature is implemented only by transition 4 in Fig. 1(d) since the corresponding product does not need to open and close the beverage compartment. Thus, this feature implementation "disagrees" with the rest. Like `require`?, this operator uses a *property* that specifies disagreements of interest and returns a *set of witnesses* exemplifying the disagreements (or none if the features agree). A simple form of disagreement is when features have different implementations, as in the above example. In that case, a *witness* could include artifacts that distinguish between the corresponding feature extensions. Disagreements can also be semantic, e.g., when checking for behavioral properties rather than the syntax of the implementing artifacts.

`interact`? determines whether combining a set of features would alter the behavior of one or more of those features. The input *property* specifies the form of interactions to be checked and the output *set of witnesses* exemplifies them. For example, a composition of features pay and free from transition systems in Figs. 1(a) and 1(d) might result in a transition system where the transition pay follows free: one has to pay after requesting a free drink, clearly violating the main behavioral property of the free feature.

`compose` combines features of the n input systems, producing a single system as a result. The *matches* parameter specifies artifacts that are considered similar and should be unified in the combined representation. In addition, the *resolution* parameter declaratively specifies how to resolve disagreements and interactions between the input features, e.g., by *overriding* one feature extension with another, *integrating* the extensions together (thus producing a "merged" implementation), or *keeping both* as separate features (with distinct FSs). For example, when composing the transition systems in Figs. 1(a) and 1(d), one might choose to override the behavior of the take drink feature in Fig. 1(d) with the one in Fig. 1(a) or keep both behaviors as *alternatives*. `compose` can be used for combining individual features (systems with a single feature each), adding a feature to an existing product (systems with a single feature combined with a system representing a well-formed product), or combining distinct products (systems each representing a well-formed product).

TABLE I
OPERATORS FOR MANAGING CLONED VARIANTS.

Operator	Input	Output
`findFS`	*product*	*set of FSs*
`findFE`	*product* × *FS* × *property*	*FE*
`require?`	<f_1, *product*> × <f_2, *product*> × *property*	*set of witnesses*
`same?`	<f_1, $product_1$> × <f_2, $product_2$> × *property*	*set of witnesses*
`interact?`	*set of <feature, product>* × *property*	*set of witnesses*
`compose`	$system_1 \times \ldots \times system_n \times$ *matches* × *resolution*	*system*

III. Scenarios

In this section, we describe two common scenarios of managing cloned product variants, exemplifying challenging software engineering activities only partially supported by existing tools and approaches. We show how these scenarios can be implemented using the operators defined in Sec. II.

Sharing Features Between Variants. Developers occasionally need to identify and share features between cloned products [6]. Fig. 3 sketches the sequence of operators supporting this activity. For example, suppose the developers of the Soda and Tea product in Fig. 1(b) decide to adapt the cancel functionality implemented by Soda with Cancel in Fig. 1(c). They retrieve the artifacts implementing cancel using `findFE`

```
Input: product_A and product_B, with product_B containing a feature of interest FS
Output: product_A with the additional feature from product_B
1.  feature = ⟨FS, findFE(product_B, FS, property_1)⟩
2.  system_A = ∅; resolution = ∅
3.  setOfFS_A = findFS(product_A)
4.  for each FS_A in setOfFS_A
5.      system_A = system_A ∪ ⟨FS_A, findFE(product_A, FS_A, property_2)⟩
6.  end for
7.  witnesses = interact? ({⟨feature_A, product_A⟩}∪⟨feature, product_B⟩, property_3)
8.  if(witnesses ≠ null) provide resolution using witnesses
9.  system_A' = compose (system_A, feature, matches, resolution)
10. return artifacts of system_A'
```

Fig. 3. Algorithm for Sharing Features.

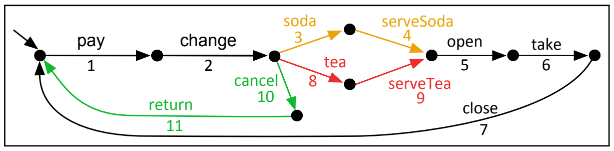


Fig. 4. Transferring the Feature Free from Product (c) to (b) in Fig. 1.

(line 1) with *property*$_1$ set to find the minimal set of artifacts directly contributing to this feature (as those will be transferred to the target product). Transitions 8 and 9 in Fig. 1(c) are returned as the result. Together with the feature specification, these transitions comprise a single-feature system.

The integration target *system*$_A$ is built similarly: first, all feature specifications of product Soda and Tea are detected using `findFS` (line 3), and then `findFE` is applied to these specifications (lines 4-6). To avoid undesired integration side-effects, `interact`? is applied on the set of all features from *system*$_A$ and the new cancel feature, w.r.t. *property*$_3$ (line 7). This property specifies behavioral characteristics of all input features and looks for violations of these characteristics as the result of feature composition. Our example does not have violations, and no resolution is needed (line 8).

Finally, `compose` combines the systems (line 9). In our example, the *matches* parameter considers artifacts to be similar if they have identical names. As the result, the option to cancel the purchase is copied from the original Soda with Cancel product to the corresponding location in the artifacts of *system*$_A$, producing the desired model in Fig. 4.

Merge-Refactoring Variants into an SPLE Representation. Another common scenario of dealing with a collection of related variants is re-engineering them into a single-copy SPL representation. The implementation of this scenario is sketched in Fig. 5: first, *system*$_i$ is built for each *product*$_i$ by detecting its FSs and the corresponding FEs (lines 1-6). Unlike Fig. 3, here `require`? is applied to each pair of features from the same product, capturing the relationships between them (lines 7-10) – this time, such relationships are an important part of the produced representation. Next, each pair of features from different products is checked for potential disagreements (lines 12-15), and all possible combinations of features are checked for potential interactions[1] (lines 16-19). If disagreements or interactions are found, developers need to provide the desired *resolution* (lines 14 and 18). The simplest resolution strategy is to *mutually exclude* the interacting features, e.g., feature

[1]Products derived from the generated SPLE representation can contain various sets of features; thus, we check all combinations. Specific detection techniques can provide optimizations to avoid a large number of checks.

```
Input: product_1 … product_n
Output: single system representing the collection of products
1.  for each product_i
2.      system_i = ∅
3.      setOfFS_i = findFS(product_i)
4.      for each FS in setOfFS_i
5.          system_i = system_i ∪ ⟨FS, findFE(product_i, FS, property_1)⟩
6.      end for
7.      for each ⟨feature_k, feature_m⟩ in S_i
8.          witnesses_1 = require? ((feature_k, product_i), (feature_m, product_i), property_2)
9.          if(witnesses_1 ≠ null) system_i = system_i ∪ (feature_k requires feature_m)
10.     end for
11. end for
12. for each ⟨feature_k, feature_m⟩ from ⟨system_k, system_m⟩ such that k ≠ m
13.     witnesses_2 = same? ((feature_k, system_k), (feature_m, system_m), property_3)
14.     if(witnesses_2 ≠ null) update resolution using witnesses_2
15. end for
16. for each set F of ⟨feature, product⟩ pairs
17.     witnesses_3 = interact? (F, property_4)
18.     if(witnesses_3 ≠ null) update resolution using witnesses_3
19. end for
20. return compose (system_1, …, system_n, matches, resolution)
```

Fig. 5. Algorithm for Merge-Refactoring Products.

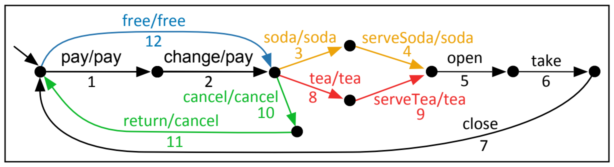


Fig. 6. Merge-Refactoring Products in Fig. 1.

free of the Free Soda product and pay of the remaining products in Fig. 1. More complex resolutions allowing to override or merge features from different products are also possible. For example, the take drink feature of the Free Soda product "disagrees" with the remaining variants: all but the Free Soda product open and close the drinks compartment after serving the drink. Thus, its behavior is replaced by the behavior of the other products. Fig. 6 shows the result of combining the products in Fig. 1 under this strategy, as produced by `compose` (line 20). Like the previous scenario, *matches* combines artifacts with the same name. The produced system has five features: pay, free, soda, tea and cancel, with the first two being alternative of each other, as stated above. The FS that corresponds to each transition in Fig. 6 is shown after the '/' character, e.g., pay corresponds to both transitions 1 and 2. The main advantages of the produced representation are lack of duplication and the ability to easily produce new product variants, e.g., a Tea with Cancel product.

IV. Operator Implementation and Remaining Gaps

In this section, we discuss existing approaches applicable for implementing the operators described in Sec. II and identify gaps to be addressed.

`findFE` (a.k.a. feature location) and `interact`? (a.k.a. feature interaction) are by far the most studied. Over 20 different feature location techniques for source code have been developed [10]. Yet, it is often unclear what the exact properties of the located feature are, how to compare techniques based on the features they detect, and how to extend these approaches to allow users to specify the desired properties of the location process. Also, feature location techniques for artifacts other than code, e.g., models, are poorly studied.

Feature interaction techniques have also received a lot of attention, especially in the telecommunications domain [11].

Most of the existing approaches, however, deal with *pairwise* feature interactions. They have to be extended to consider interactions between sets of features that are part of real-life products: such sets can introduce interactions that are not detectable in a pairwise manner. Also, the applicability of many techniques for analyzing feature interactions is limited because they are designed to work on special-purpose models rather than production artifacts.

Compare and *merge* techniques, for both code and models [12], [13], as well as aspect weaving [14] and feature-oriented composition approaches [15], can be used to realize `compose`. Such techniques need to be extended to allow specifying the desired *resolutions*. Also, the techniques should be able to deal with unstructured product slices that correspond to feature extensions rather than complete, well-formed products or features declared in a specific manner. Lastly, approaches considering the "global" picture and devising strategies for combining n inputs simultaneously, rather than doing so in a pairwise manner, are to be developed.

Syntactic and semantic comparison techniques [16], [17] can also be used to implement the operator `same`?. However, future work is required to adapt them for analyzing unstructured feature extensions and declaratively obtaining the desired properties of the analysis.

Code analysis techniques, e.g., *program slicing* [18], can be used to implement `require`?. Similar techniques are not well developed for artifacts other than code. Also, it should be possible to explicitly classify the techniques based on the nature of the detected `require` relationships and even retrieve `require` relationships of a desired type.

Finally, implementations of the `findFS` operator do not seem to exist. Such implementations should take into account a variety of product artifacts, including product documentation, marketing reports, etc. Techniques for decoupling product code, such as [19], and then concisely summarizing each part individually, as in [20], could be extended to produce FSs.

V. Summary and Related Work

In this paper, we focused on the problem of managing a collection of related product variants realized via cloning – a practice commonly taken in industry. Despite numerous isolated solutions aiming to assist the practitioners in managing such variants, this task is still challenging. We thus took a systematic, top-down approach for identifying a set of required operators and demonstrated their applicability for realizing two real-life scenarios. Our operators are intentionally abstract; moreover, the presented set might not be complete and some of the operators we considered atomic might need to be further split up. Yet our work provides a first step towards specifying and organizing solutions required for dealing with cloned variants. We discussed possible strategies for implementing the proposed operators, including synergies with solutions outside the SPLE domain, and identified the remaining gaps. Based on those, we hope to be joined by the research community in devising dedicated solutions and empirically studying their applicability.

Related work. Other authors also looked at systematic classifications of programming tasks: Chen and Rajlich [9] identified six fundamental program comprehension operators that trace feature label, description and extension to each other. We incorporated some of them in our work, however, our main focus was on cases of *multiple* variants rather than single-copy systems and involved *manipulations* on the variants rather than only comprehension activities. Borba et al. [21] suggest a theory of product line refinement. This is a special case of variant management, the problem we consider here. Brunet et al. [22] identified model merging operators and specified their algebraic properties. Our work is not limited to models and considers a broader set of necessary maintenance activities.

References

[1] P. C. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[2] T. Mende, R. Koschke, and F. Beckwermert, "An Evaluation of Code Similarity Identification for the Grow-and-Prune Model," *J. of Soft. Maintenance and Evolution*, vol. 21, no. 2, pp. 143–169, 2009.

[3] J. Rubin and M. Chechik, "Combining Related Products into Product Lines," in *Proc. of FASE'12*, 2012, pp. 285–300.

[4] J. van Gurp and C. Prehofer, "Version Management Tools as a Basis for Integrating Product Derivation and Software Product Families," in *Proc. of VaMoS'06*, 2006, pp. 48–58.

[5] C. Thao, E. Munson, and T. Nguyen, "Software Configuration Management for Product Derivation in Software Product Families," in *Proc. ECBS'08*, 2008, pp. 265–274.

[6] 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. of CSMR'13*, 2013.

[7] J. Rubin, K. Czarnecki, and M. Chechik, "Managing Cloned Variants: A Framework and Experience," 2013, submitted.

[8] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin, "Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines," in *Proc. of ICSE'10*, 2010.

[9] K. Chen and V. Rajlich, "Case Study of Feature Location Using Dependence Graph," in *Proc. of IWPC'00*, 2000, pp. 241–249.

[10] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature Location in Source Code: A Taxonomy and Survey," *J. of Soft. Maintenance and Evolution*, vol. 23, no. 8, 2011.

[11] P. Zave, "FAQ Sheet on Feature Interaction," http://www2.research.att.com/~pamela/faq.html, 2004.

[12] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall, "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction," *IEEE TSE*, vol. 33, pp. 725–743, 2007.

[13] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave, "Matching and Merging of Statecharts Specifications," in *Proc. of ICSE'07*, 2007, pp. 54–64.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," in *ECOOP'01*, 2001.

[15] D. S. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling Step-Wise Refinement," *IEEE TSE*, vol. 30, no. 6, pp. 355–371, 2004.

[16] S. Horwitz, "Identifying the Semantic and Textual Differences Between Two Versions of a Program," in *Proc. of PLDI'90*, 1990, pp. 234–245.

[17] D. Jackson and D. A. Ladd, "Semantic Diff: A Tool for Summarizing the Effects of Modifications," in *Proc. of ICSM'94*, 1994, pp. 243–252.

[18] F. Tip, "A Survey of Program Slicing Techniques," *J. Prog. Lang.*, vol. 3, no. 3, 1995.

[19] K. Herzig and A. Zeller, "Untangling changes," Sep. 2011, manuscript.

[20] S. Rastkar, G. C. Murphy, and A. W. J. Bradley, "Generating Natural Language Summaries for Crosscutting Source Code Concerns," in *Proc. of ICSM'11*, 2011, pp. 103–112.

[21] P. Borba, L. Teixeira, and R. Gheyi, "A Theory of Software Product Line Refinement," *TCS*, vol. 455, pp. 2–30, 2012.

[22] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh, "A Manifesto for Model Merging," in *Proc. of GaMMa'06*, 2006, pp. 5–12.