Clone-and-Own Software Product Derivation Based on Developer Preferences and Cost Estimation

Eddy Ghabach, Mireille Blay-Fornarino, Franjieh El Khoury, Badih Baz

Clone-and-Own Software Product Derivation Based on Developer Preferences and Cost Estimation

Eddy Ghabach and Mireille Blay-Fornarino
Université Cote d'Azur
I3S, CNRS UMR 7271
Sophia Antipolis, France
{ghabach,blay}@i3s.unice.fr

Franjieh El Khoury
Université de Lyon 1
ERIC EA 3083
Lyon, France
franjieh.elkhoury@eric.univ-lyon2.fr

Badih Baz
Université Saint-Esprit de Kaslik
Kaslik, Lebanon
badih.baz@usek.edu.lb

***Abstract*—Clone-and-own is a common reuse practice that is widely adopted for evolving a family of software systems. However, this practice loses its effectiveness if not supported with valuable indicators that guide the derivation of new products. In this paper, we propose an approach to support the derivation of new product variants based on clone-and-own, by providing the possible scenarios in terms of operations to perform to accomplish the derivation. We generate a constraints system prior to a product derivation, to facilitate the software engineer selection of the suitable scenario and operations based on his preferences. In addition, we propose a cost estimation for each operation and respectively for each scenario, thus, a software engineer can rely on it as an additional parameter to achieve the derivation. The proposed scenarios and cost estimation are based on indicators retrieved after an automated identification of the mappings between the features implemented by the family of software products and the assets in which they are implemented. We preliminarily validate our approach on a case study where results show that the provided support can considerably reduce the amount of time and efforts that can be required to achieve a product derivation.**

***Index Terms*—Clone-and-own, product derivation, software reuse.**

I. Introduction

Establishing a family of software systems is done usually either by adopting software product line engineering as a top-down approach to construct a software product line (SPL) [1] or by adopting an ad-hoc practice such as copy-paste-modify or clone-and-own [2], [3]. SPLs proven their success in providing a systematic reuse [4], [5] by reducing development costs and time to market, and increasing software quality [6]. Regardless their impressive return on investment [6], SPLs are considered as an expensive up-front investment [7] that consists of initially defining reusable artifacts in a domain engineering phase, before deriving new products through an application engineering phase [6]. Hence, SPLs are often established subsequently to the development of several software variants using simple and rapid ac-hoc practices [8], [9], [10]. Several works in literature propose approaches to migrate similar software variants into an SPL [11], [12], [13], [14].

Evolving a family of software products consists often in deriving new variants by reusing the existing ones. Despite that SPLs provide systematic reuse due to variability management, product derivation is restricted to the product line portfolio. Hence, deriving new products consists of evolving the SPL at both domain and application engineering levels, a task that is considered complex due to variability and interdependency between products [15]. On the other hand, clone-and-own (C&O) is a common intuitive practice that consists in cloning an existing product variant (PV) then modifying it to add and/or remove some functionalities in order to derive a new PV [10], [16], [17]. C&O is characterized by the availability, rapidity and simplicity of cloning an existing PV into a new one ready for modification, in addition to the freedom that a software engineer can benefit from to achieve product derivation [3]. Although being a time and cost saving practice, C&O might turn into an expensive and inefficient solution if tracking about the artifacts existing in several clones is lacked, which produces an incertitude in identifying the PV(s) to be considered as source for cloning [3], [10].

In this paper, we address the derivation of new PVs from a family of software products, where C&O is adopted to achieve the derivation, regardless if it is performed in an SPL context or not. We are interested in supporting the derivation without automating it or imposing a specific solution to achieve it. We aim to guide a software engineer to achieve the derivation on his own, in order to preserve the "own" side of the C&O practice, where the software engineer is the decision maker.

A PV is composed of a set of files a.k.a assets, and it implements a set of business functionalities a.k.a features, where the PVs that belong to the same family of software systems share features in common [18]. The derivation of a new PV is needed when no PV implements all and only the requested features. Yet, the question is: *what are the information and indicators that can be investigated to determine the suitable reuse scenario to derive the requested PV?* Given a family of 3 PVs[1] shown in Table I, to deliver a product that allows to *manage*, *add* and *delete* matches, a new PV – say p_4 – has to be derived, since no PV implements all and only the requested features. Thus, based on what a software engineer can decide to derive p_4 either by (1) cloning p_2 and removing from the clone the code fragments related to $ModifyMatches$, or (2) by cloning p_3 and extracting the code fragments related to

[1]The implementation files of the product variants are available on: https://github.com/eddyghabachi3s/SoccerManager

$DeleteMatches$ from p_2 and integrate it in the clone? Hence, a software engineer needs answers to the following questions in order to decide which scenario is the suitable one to achieve the derivation:

Q1: What are the existing PVs that implement the features required for a derivation? and consequently, what are the combinations of PVs that constitute each possible scenario?

Q2: What are the assets of the existing PVs that implement a certain feature?

Q3: What are and how many are the assets that have to be cloned and modified for each possible scenario (combination of PVs)? and consequently, what might be the cost to perform each of these asset level operations? Respectively, which scenario provides the least expensive derivation cost?

In this paper, we propose an approach that responds to the above questions by providing the following:

1) An automated and incremental technique to identify (and update) mappings between the features implemented by the PVs and the assets used to implement them.
2) An automated method to determine the possible scenarios to achieve the derivation using C&O.
3) An auto-generated constraints system and cost estimation for the operations to perform, guiding software engineers to construct a derivation scenario based on their own preferences and the provided cost estimation.

TABLE I
PRODUCT VARIANTS WITH THE FEATURES THEY IMPLEMENT

		Products		
		p_1	p_2	p_3
Features	$ManageMatches$	✓	✓	✓
	$AddMatches$	✓	✓	✓
	$ModifyMatches$	✓	✓	
	$DeleteMatches$		✓	

II. Approach Overview

Given a set of software products $\mathcal{P} = \{p_1, ..., p_x\}$, the set of features *implemented* by $\mathcal{P}$ is $\mathcal{F} = \{f_1, ..., f_y\}$ and the set of assets *employed* by $\mathcal{P}$ is $\mathcal{A} = \{a_1, ..., a_z\}$. We denote $F(p_j)$ the features implemented by a certain product p_j, and $A(p_j)$ the assets employed by p_j. An asset represents an implementation file, and since each product might *exploit* a specific version of the file, we refer to file versions as *asset instances*. Thus, for each asset $a_k \in A(p_j)$, p_j *exploits* one of its instances a_k^i. We denote $AI(a_k)$ the instances of a_k, and $AI(p_j)$ the asset instances exploited by p_j to fulfill its implementation.

A. Mapping features to assets

To determine what are the assets that contribute in the implementation of a certain feature, mappings between the features $\mathcal{F}$ and the assets $\mathcal{A}$ must be identified. Ziadi *et al.* propose an approach to identify the features implemented by a family of products in case they are not identified [9]. Feature identification is achieved by analyzing the artifacts of the products. In our approach, we are not interested in feature identification, since we consider that the features implemented by each product are determined by domain experts prior to its implementation. Hence, at this level, we focus only on identifying mappings between features and assets. Several works in literature employ feature location techniques that analyze the co-occurrence between features and assets to identify mappings between them [11], [12], [16]. We propose in this paper, a simple and automated technique to identify mappings between features and assets for polyglot systems, that in contrary to several existing approaches, is independent of the artifacts type, and treats the products as a family of related entities and not as individual independent entities [19].

In order to identify mappings, we define *"correlations"*. A *correlation*[2] indicates the coexistence between a feature and an asset, or between a feature and an asset instance. An asset is a global abstraction of a partial or a total implementation of one or more features, since a feature implementation can be spread into several assets, and similarly, an asset can include implementation fragments of different features. On the other hand, an asset instance realizes one of the asset implementations in one or more products. Instead of mapping a feature or set of features (features interaction) to an implementation block which can be composed of fragments of several assets, we map each feature to the set of assets (atomic files) that supposedly contribute in its implementation. Hence, a feature might be correlated to several assets, and an asset might be correlated to several features as well.

A correlation between a feature f and an asset a holds if the following constraints are valid:

- For each product p_j that implements f, p_j employs a (*idem* exploits any of its instances).
- There does not exist an asset instance a^{i} exploited by p_j and by another product p_k that does not implement f.

Thus, given $P(f)$ the set of products that implement f and $P(a)$ the set of products that employ a, a correlation between f and a denoted as $c(f, a)$ holds, if $P(f) \subset P(a) \wedge \forall a^{\text{i}} \in AI(P(f)), a^{\text{i}} \notin AI(P(a) \setminus P(f))$.

Respectively, given an instance a^{i} of an asset a, a correlation between a feature f and a^{i} denoted as $c(f, a^{\text{i}})$ holds if $c(f, a) \wedge \exists p, f \in F(p), a^{\text{i}} \in AI(p)$.

To guarantee that a feature, asset or asset instance participates at least in one correlation, we impose the following rules:

1) No two products implement exactly the same set of features.
2) There must exist at least one common feature between all products.
3) If a product implements all features implemented by another product and more, the former product must employ at least all the assets employed by the latter product.

[2]Correlation: a mutual relationship or connection between two or more things (Oxford Dictionary)

B. Determining possible scenarios and operations to achieve a derivation

Reusing artifacts of existing PVs is essential to achieve the derivation of a new PV. This derivation might introduce new features that were not implemented earlier by existing PVs. We define a *configuration* cf as a pair composed of two sets of features: $EF(f)$ the existing required features where $EF(cf) \subseteq \mathcal{F}$, and $NF(cf)$ the new required features where $NF(cf) \not\subset \mathcal{F}$. Thus, having cf_4 the configuration relative to the derivation of p_4, $EF(cf_4) = \{ManageMatches, AddMatches, DeleteMatches\}$, while $NF(cf_4) = \{\phi\}$. During derivation, if $NF(cf) \neq \{\phi\}$, our approach does not provide any guidance concerning which assets are suspect to modification to integrate the new features. For instance, if $NF(cf) = \{StatMatches\}$ where $StatMatches$ is a required feature to display the statistical details of a match, software engineers have to determine the assets to be added and/or modified to introduce this feature in the product to derive.

Several products or combinations of products can be reused to achieve the derivation of a new PV. Hence, for a given configuration cf, we define *configuration scenarios* $CS(cf)$ that represent the set of all possible combinations of products that can achieve the configuration. We define a *configuration scenario* denoted $cs_i(cf)$ as a pair $\langle\{\langle p_k, \{f_q, ..., f_s\}\rangle\}, \{f_x, ..., f_z\}\rangle$, where $\{\langle p_k, \{f_q, ..., f_s\}\rangle\}$ is a combination of products that can be reused to achieve the configuration and $\{f_x, ..., f_z\}$ is $NF(cf)$ if any. A product is candidate for a configuration scenario if it implements at least one of the features of $EF(cf)$. Further, for each combination, the unrequired features $\{f_q, ..., f_s\}$ implemented by a candidate product p_k are identified.

In order to be able to determine the suitable configuration scenario, a software engineer aims to have further information about the operations to perform in each scenario to achieve the derivation. Such information must involve the identification of the assets required from the products of each configuration scenario, in addition to the operations to perform at the level of each asset in order to construct the required product. Given a product p, an asset a employed by p is required for the derivation if $F(a) \bigcap EF(cf) \neq \{\phi\}$, where $F(a)$ are the features that a is in correlation with. Hence, for each required asset, one of its instances has to be cloned, then modified if necessary to remove implementation fragments corresponding to unrequired features, and if there still exist some required features that are not implemented by the cloned instance, their implementation fragments must be extracted from the other instances of the asset and integrated in the clone. The resulting asset instance of each required asset has to implement the set of features $F(a) \bigcap EF(cf)$. Thus, we identify three types of actions that might be taken over the instances of a required asset in order to produce the desired instance:

1) Clone and Retain (CRT): clone an asset instance and retain it as it is, without modifying its implementation.
2) Clone and Remove (CRM): clone an asset instance, and remove from it the implementation fragments corresponding to the features that it is in correlation with but are not required by the configuration.
3) Extract and Add (ETA): extract from an asset instance the implementation fragments of some features required by the configuration, and add them to a cloned instance under construction. An ETA action is used only as a subsequent to a CRT or CRM action in order to complete the construction of a cloned instance with extracted implementation fragments.

An *action* ac is defined as a triple $\langle type, a^{\mathrm{i}}, \{f_j, ..., f_n\}\rangle$, where $type$ corresponds to one of the types defined above: $\{CRT, CRM, ETA\}$. For CRT and CRM actions, a^{i} corresponds to the asset instance to clone. For an ETA action, a^{i} corresponds to an asset instance to extract from. Whereas, $\{f_j, ..., f_n\}$ corresponds to the set of features to remove from a^{i} if the action is CRM, or to extract from a^{i} if the action is ETA. Hence, the resulting asset instance for a required asset is produced by cloning an asset instance exploited by a product of the configuration scenario using a CRT or CRM action, removing the implementation fragments corresponding to the unrequired features in case of a CRM action, and extracting the remaining required features from other instances using an ETA action, if any.

We define an *operation* as the set of actions needed to produce the desired asset instance. Thus, an operation op is a triple $\langle a, \{ac_1, ..., ac_n\}, a^{\mathrm{i}}\rangle$ where a is the required asset, and $\{ac_1, ..., ac_n\}$ noted as $AC(op)$ is the set of actions to be made to obtain the desired asset instance a^{i}. Given an asset $style.css$, the operation: $\langle style.css, \{\langle CRM, style.css^1, \{ModifyMatches\}\rangle, \langle ETA, style.css^2, \{DeleteMatches\}\rangle\}, style.css^4\rangle$ consists of cloning the asset instance $style.css^1$ and removing from it the feature $ModifyMatches$, then extracting the feature $DeleteMatches$ from $style.css^2$ and adding the extraction to the clone, which produces a new instance $style.css^4$.

For each configuration scenario, at the level of each asset, several operations might be possible to construct the desired instance of the asset. Therefore, to achieve the derivation of a new PV, a software engineer is provided with all possible configuration scenarios, and all possible operations that can be performed at asset level. We call *derivation scenario* the complete set of operations that a software engineer selects in order to accomplish the derivation of a new PV.

C. Derivation based on developer preferences and cost estimation

It is quite useful to support the derivation of a new PV with all possible scenarios and operations. However, in case of a large family of software products where assets might have a large number of instances, the number of possible scenarios and operations are supposed to become very large. Therefore, if the provided support is not accompanied with valuable indicators it might lose its relevance. For this reason, we strengthen the support provided by our approach with two factors.

The first factor is to allow software engineers to select the suitable scenario or operations to perform based on their personal preferences. To do so, based on the identified configuration scenarios and operations, we construct a *constraints system* by means of a feature model, that allows a software engineer to choose a derivation scenario from several dimensions within or outside the context of a configuration scenario. The features of the generated feature model are the configuration scenarios, their products, their operations and their corresponding assets and asset instances, while the constraints correspond to the dependencies between them. From the first dimension, a software engineer can select the configuration scenario that requires the least number of operations that impose a construction of a new asset instance, or the configuration scenario that is composed of the products that she is most familiar with. From a second dimension, the constraints system allows to filter the operations based on the deselection of some undesirable products. From a third dimension, it allows to filter operations by deselecting undesirable instances of assets whenever possible, such as old or untrusted instances. Thus, the constraints system allows not only to take the software engineer preferences into consideration, but also to reduce the number of decisions to be taken which simplifies the construction of the derivation scenario.

The second factor is to provide an estimated cost in terms of development effort and time for each operation and respectively for each configuration scenario. The cost of an operation has to be estimated based on the actions that it is composed of. In case of a CRT action, an asset instance has to be cloned without being modified, therefore, no cost has to be allocated. In case of a CRM action, an asset instance has to be cloned and implementation fragments corresponding to one or more features must be removed from the cloned instance. Similarly, for an ETA action, implementation fragments corresponding to one or more features must be extracted from an instance and integrated in the cloned instance. Therefore, for both CRM and ETA a cost has to be allocated. We assume that an ETA action costs 50% addition efforts compared to a CRM action, since it consists in adding the extracted fragments to the clone after their extraction. Thus, we define an *action type weight* denoted aw, where $aw = 0$ for CRT, $aw = 1$ for CRM and $aw = 1.5$ for ETA. We estimate the cost of removing or extracting a feature based on the following global assumption: *as much as the correlation degree between a feature and an asset instance is high, the removal or extraction of the feature from the asset instance becomes hard*. We define *correlation degree* based on the following assumptions:

- *As much as the number of features that an asset instance is in correlation with increases, the correlation degree between the asset instance and any of those features decreases*. The features that an asset instance a^{i} is in correlation with are denoted as $F(a^{\mathrm{i}})$. Hence, a feature $f \in F(a^{\mathrm{i}})$ corresponds to $1 \div |F(a^{\mathrm{i}})|$.
- *As much as the number of assets that a feature is in correlation with increases, the correlation degree between the feature and any of those assets decreases*. The assets that a feature f is in correlation with are denoted as $A(f)$. Hence, an asset $a \in A(f)$ corresponds to $1 \div |A(f)|$.
- *As much as the number of instances of an asset that a feature is in correlation with increases, in relation to the overall number of instances of the asset, the correlation degree between the feature and the asset increases*. The instances of an asset a that a feature f is in correlation with are $AI(a) \cap AI(f)$. Hence, the number of instances of a that f is in correlation with, in relation to the overall number of instances of a corresponds to $|AI(a) \cap AI(f)| \div |AI(a)|$.

We define a *correlation degree* between a feature f and an asset instance a^{i} as:

$$cd(f, a^{\mathrm{i}}) = \frac{1}{|F(a^{\mathrm{i}})|} \times \frac{1}{|A(f)|} \times \frac{|AI(a) \cap AI(f)|}{|AI(a)|}$$

Thus, the cost of an action is the sum of the correlation degrees of the features that have to be removed or extracted from the asset instance of the action which are multiplied by the *action type weight* aw.

$$cost(ac) = \sum_{f_j}^{f_n} (cd(f_j, a^{\mathrm{i}}) \times aw)$$

Respectively, the cost of an operation is the sum of the cost of all its actions.

$$cost(op) = \sum_{i=1}^{n} cost(ac_i)$$

Providing the estimated cost of the operations facilitates the selection of an operation when several operations are possible to construct an asset instance. Moreover, we consider the cost of a configuration scenario as the sum of the estimated cost of the operations having the lowest cost at each asset level. Therefore, a software engineer can rely on the estimated cost of the configuration scenarios as an additional parameter during the construction of a derivation scenario. It is important to mention that the estimated cost of removing or extracting a feature from an asset instance does not necessarily assert that some implementation fragments have to be removed or extracted. We consider the estimated cost as the time and development effort that must be achieved in order to revise the asset instance source code, and determine if there exists some source code related to the feature to remove or extract, and if so, accomplish the task and test the cloned instance.

D. Sustainable evolution

Software engineers aim not only to acquire a support to derive new PVs from a family of software systems, but also to benefit from the newly derived PVs as additional elements of support during the derivation. Our technique to identify correlations is incremental, allowing the identification of new correlations whenever new features, assets or asset instances are added. In addition, it breaks existing correlations in case they are not valid anymore. Hence, the identification of a

new PV implies an automated and incremental update of the correlations. The evolution of the correlations reinforces reuse and contributes in a sustainable evolution of the family of software products.

III. Preliminary Validation

We preliminarily validated our approach on a real case study that we developed for this purpose. The case study consists of 8 variants of a web application. The family of products comprises when it has its 8 variants a total of 93 features, 271 assets and 296 asset instances with an average of 66 features, 214 assets and 4.7KLOCs per variant. We achieved the validation by analyzing statistical information that we collected upon the incremental derivation of 5 variants from an initial family of 3 variants. By incremental derivation we mean that correlations are updated subsequently to the derivation of a new PV that becomes a support element during the upcoming derivation. Information about the features, assets and asset instances added through derivations are show in Table II.

As shown in Fig 1 the average number of configuration scenarios per configuration increases considerably whenever the family of software products becomes richer. Respectively, Fig 2 shows that the average number of products involved in a configuration scenario increases too. These indicators reveal the importance of our approach in providing a cost estimation for the configuration scenarios, as well as filtering configuration scenarios based on preferences over products. Fig 3 shows that the average number of assets to modify decreases whenever the family of products becomes mature. The strength of our approach here is that it determines and separates the operations requiring modifications (construction of a new instance of an asset), from the ones that require only a clone of one of the instances of an asset without modifying the clone. This fact is manifested in the cost estimation of the operations. The collected statistical information show also in Fig 4 that the coefficient of variation between the estimated cost of the configuration scenarios is elevated. The coefficient of variation that ranges between 36% and 99% with an average of 64% reflects that if software engineers select the configuration scenarios having the lowest estimated cost, they can save a considerable amount of time and effort.

TABLE II
METRICS OF 5 CONFIGURATIONS TO DERIVE NEW PVS

Configuration	cf_4	cf_5	cf_6	cf_7	cf_8
Added features	0	0	25	0	0
Added assets	0	0	8	0	24
Added asset instances	3	1	11	1	25

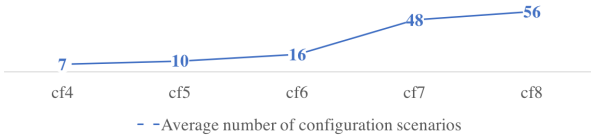

Fig. 1. Average number of configuration scenarios per configuration.

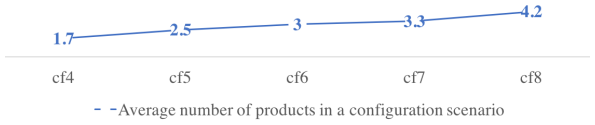

Fig. 2. Average number of products involved in a configuration scenarios per configuration.

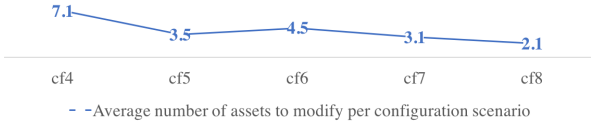

Fig. 3. Average number of assets to modify per configuration scenario per configuration.

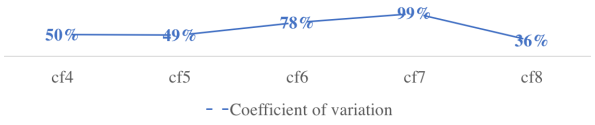

Fig. 4. Coefficient of variation between the estimated cost of the configuration scenarios per configuration.

IV. Limitations and Threats to Validity

1) Limitations: The correlations identification in our approach is dependent on the structural architecture of the software variants. A modification of the structure or the name of a certain artifact affects the identified correlations. Another limitation consists in the correlations that are identified at file level whereas several related works [12], [16] map features to implementation blocks of several files. These approaches could be complementary to our work, however, we consider that dealing with polyglot systems such as web applications necessitates to provide support at file level.

2) Threats to validity: The validation of our approach on different case studies might produce different results depending on the architecture of the product variants in concern and the propagation of the features in the product variants and respectively in their assets. For instance, we might face a family of products where the average number of assets to modify remains high when the family of products becomes richer. However, the metrics that we rely on to identify correlations or either estimate costs are generalized metrics that are supposed to be correct regardless the architecture of the software products. For the preliminary validation case study, the order between estimated costs and actual workload corresponds to those in practice, however, this does not guarantee that this assumption is correct for other case studies.

V. Related Work

Lapeña *et al.* [17], [20] propose an approach called *CACAO* to assist C&O. When a new product has to be derived, the requested documented requirements are provided. *CACAO*

extracts keywords from the requested requirements and from the existing product variants requirements and detects which existing product variants are closer to the product variant to derive in terms of requirements. Finally, it determines which are the source code methods of the existing product variants that are closer to the requested requirements. Hence, *CACAO* provides ranking at products level and methods level. The *CACAO* approach is very relevant to our work, despite that in our approach we give interest in providing additional operational support by proposing the detailed operations to perform to achieve the derivation. A framework for enhancing C&O with systematic reuse called *ECCO* is proposed by Fischer *et al.* [16], [21]. *ECCO* integrates software variants to provide a systematic reuse of the existing PVs. Further it automates the derivation of new PVs, and if the derivation requires a manual completion, it proposes the necessary hints for completion. On the contrary, our approach provides the possible operations to perform on the required assets, allowing software engineers to complete the derivation based on their own preferences, preserving the meaning of C&O which gives software engineers the freedom to derive products on their own. Martinez *et al.* [7], [12] propose an approach called *BUT4Reuse* that allows the integration of software variants into an SPL by constructing its feature model. *BUT4Reuse* provides an automated derivation of existing and new variants. However, it does not permit an incremental evolution of the SPL. Similarly, AL-Msie'Deen *et al.* [11] propose an approach to mine features and construct a feature model from product variants. However, the approach is limited to object-oriented based variants. Rubin and Chechik, suggest a framework to manage PVs developed using C&O approach [14] and they define a set of useful operators to manage PVs and derive new ones. Some of those operators are provided by our approach, while others can be integrated if needed.

VI. Conclusion and Future Work

In this paper, we proposed an approach to support the derivation of new product variants from existing ones using clone-and-own practice. First, we defined an automated technique to identify correlations, which are the mappings between features and assets of the product variants. Second, to support the product derivation, we employ the correlations to determine the possible configuration scenarios that can lead to the derivation of the new product, in addition to the operations to perform at asset level for each scenario. We enhance the provided support with constraints system allowing software engineers to select the suitable scenario and operations based on their own preferences, in addition to a cost estimation of the operations to facilitate their selection. We preliminarily validated our approach on a case study developed for this purpose, where results show that our approach can save a considerable amount of time and effort during product derivation. As future work, we aim to validate the effectiveness of our approach by testing it on more sophisticated systems of different architectures. Moreover, we are interested in identifying additional metrics to refine the cost estimation function [22] by taking organizational factors in concern [23]. Finally, we aim to integrate our approach in a global software product line environment [24] to benefit from systematic reuse of existing products and variability management during configuration.

References

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

[2] G. Zhang et al., "Cloning practices: Why developers clone and what can be changed", ICSM, Italy, 2012, pp. 285–294

[3] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, K. Czarnecki, "An exploratory study of cloning in industrial software product lines" CSMR, Genova, Italy, Mar 2013, pp. 25–34

[4] S. Daniel and T Eisenbarth, "Evolutionary introduction of software product lines", SPLC, San Diego, CA, USA, Aug 2002, pp. 272–283

[5] J. Bosch, "Software product families: towards compositionality", *Fundamental Approaches to Software Engineering*, Springer, 2007, pp. 1-10

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

[7] J. Martinez, "Mining software artefact variants for product line migration and analysis", *PhD*, Université Pierre et Marie Curie - Paris VI, Oct 2016

[8] H. Eyal-Salman, A. Seriai, C. Dony, R. Al-msie'Deen, "Recovering Traceability Links Between Feature Models and Source Code of Product Variants", in *Proceedings of the VARiability for You Workshop: Variability Modeling Made Useful for Everyone*, Austria, Sep 2012, pp. 21–25

[9] T. Ziadi et al., "Feature Identification from the Source Code of Product Variants", CSMR, Hungary, 2012, pp. 417–422

[10] L. Linsbauer, S. Fischer, R. E. Lopez-Herrejon, and A. Egyed, "Using Traceability for Incremental Construction and Evolution of Software Product Portfolios", in *8th International Symposium on Software and Systems Traceability*, IEEE/ACM, Florence, Italy, May 2015, pp. 57-60

[11] R. AL-MsieDeen, "Reverse Engineering Feature Models from Software Variants to Build Software Product Lines", *PhD*, Montpellier, 2014

[12] 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", SPLC, USA, ACM, July 2015, pp. 101-110

[13] 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 *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, Gyeongju, Korea, ACM, Mar 2014, pp. 1064–1071

[14] J. Rubin, K. Czarnecki, and M. Chechik, "Managing cloned variants: a framework and experience", SPLC, ACM, Aug 2013, pp. 101–110

[15] G. Botterweck, and A. Pleuss, "Evolution of Software Product Lines", *Evolving Software Systems*, Springer, Berlin, 2014, pp. 265–295

[16] S. Fischer, L. Linsbauer, R.E. Lopez-Herrejon and A. Egyed, "Enhancing clone-and-own with systematic reuse for developing software variants", ICSME, Canada, Sep 2014, pp. 391–400

[17] R. Lapeña, M. Ballarin and C. Cetina, "Towards Clone-and-own Support: Locating Relevant Methods in Legacy Products," SPLC, Beijing, China, Sep 2016, pp. 194–203

[18] T. Berger et al., "What is a Feature?: A Qualitative Study of Features in Industrial Software Product Lines", SPLC, USA, 2015, pp.16–25

[19] J. Rubin and M. Chechik, "A survey of feature location techniques", *Domain Engineering*, Springer, 2013, pp. 29–58

[20] R. Lapeña, J. Font, C. Cetina, O. Pastor, "Model Fragment Reuse Driven by Requirements", in *Proceedings of the Forum and Doctoral Consortium Papers Presented at the 29th International Conference on Advanced Information Systems Engineering*, Esse, Germany, June 2017, pp. 12–16

[21] S. Fischer, L. Linsbauer, R.E. Lopez-Herrejon, A. Egyed, "The ECCO tool: Extraction and composition for clone-and-own", ICSE, IEEE Press, Volume 2, 2015, pp. 665–668

[22] A. Magazinius, S. Börjesson, R. Feldt, "Investigating intentional distortions in software cost estimation–An exploratory study", in *Journal of Systems and Software*, Elsevier, 85(5), 2012, pp. 1770–1781

[23] D. Badampudi, et al., "A decision-making process-line for selection of software asset origins and components", in *Journal of Systems and Software*, Elsevier, 135, 2018, pp. 88–104

[24] E. Ghabach, M. Blay-Fornarino. F. El Khoury, B. Baz, "Guiding Clone-and-Own When Creating Unplanned Products From a Software Product Line", ICSR, Madrid, Spain, 2018