Guiding Clone-and-Own When Creating Unplanned Products from a Software Product Line

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

Guiding Clone-and-Own when Creating Unplanned Products from a Software Product Line

Eddy Ghabach[1,2], Mireille Blay-Fornarino[1], Franjieh El Khoury[2,3], and Badih Baz[2]

[1] Université Côte d'Azur, I3S, CNRS UMR 7271, Sophia Antipolis, France
{ghabach,blay}@i3s.unice.fr
[2] Université Saint-Esprit de Kaslik, Kaslik, Lebanon
badih.baz@usek.edu.lb
[3] Université de Lyon, Université Lyon 1, ERIC EA 3083, Lyon, France
franjieh.elkhoury@eric.univ-lyon2.fr

Abstract. Clone-and-own is a simple and intuitive practice adopted to construct new product variants based on existing ones. However, when the developed family of products becomes rich, maintaining shared assets and managing variability between the clones become tedious tasks. Therefore, migrating the family of products into a software product line becomes essential. Despite that, software engineers remain interested in constructing new product variants that are not provided by the software product line. In this short paper, we briefly present our approach to guide software engineers in deriving new products from a software product line based on clone-and-own. This approach consists of proposing the possible configuration scenarios by means of operations to perform at asset level, in order to derive a new product variant.

Keywords: Software product line · Clone-and-own · Product derivation

1 Introduction

A software product line (SPL) is a set of software products that belong to the same domain and have some characteristics in common [1]. These characteristics are known as features [2]. A feature model (FM) is one of the abstract representations of SPL products variability [3]. A configuration is a selection of features that respects the constraints imposed by the FM and generally reflects a product of the SPL [4]. SPLs permit a systematic reuse of software artifacts, which reduces development cost and increases time to market and software quality [5]. SPLs are considered as an expensive up-front investment, since artifacts must be initially defined in a domain engineering phase, before deriving new products through an application engineering phase [5]. Therefore, organizations that are not able to deal with such an up-front investment, tend to develop a family of software products using simple and intuitive practices such as clone-and-own.

Clone-and-own (C&O) is an approach that consists in cloning an existing product variant (PV) then modifying it to add and/or remove some functionalities in order to obtain a new PV [6] [7]. This approach is practically adopted by several organizations as "favorable and natural" solution to develop a family of related software systems, due to its simplicity, availability and rapidity [8]. However, when the number of variants increases, it becomes difficult to manage them efficiently [8]. Thus, it becomes essential to migrate the developed PVs into an SPL [9], in order to manage their variability and benefit from a systematic reuse. This process is known as extractive [10] or bottom-up [11] adoption, or re-engineering [9] [12] of SPLs. In our approach, we are interested in organizations that adopt C&O to develop a family of software products, and desire to develop new PVs after integrating the existing products into an SPL. Such organizations are in need of a guidance in reusing the existing products artifacts to derive the new "desired product". Hence, our approach consists of proposing the possible configuration scenarios by means of operations to perform, in order to derive a new PV from the SPL based on C&O. In addition, our approach allows the integration of the newly developed products into the SPL, in order to benefit from its reuse in future derivations.

2 Approach Overview

We illustrate our approach on a running example, representing an excerpt of three PVs for managing soccer matches[1]. The features, assets and FM of the running example are illustrated in Table 1, Table 2, and Fig. 1 respectively.

2.1 $\mathcal{SPL}$ definition and correlations

We define $\mathcal{SPL}$ as a software product line, where $\mathcal{P} = \{p_1, ..., p_x\}$ is the set of products that can be derived through the valid configurations of the $\mathcal{SPL}$ feature model, $\mathcal{F} = \{f_1, ..., f_y\}$ is the set of features *implemented* by its products and $\mathcal{A} = \{a_1, ..., a_z\}$ is the set of assets *employed* by the products to implement the features. We note $F(p_j)$ the set of features implemented by a product p_j. A product p_j *employs* a set of assets $A(p_j)$ and for each employed asset $a_k \in A(p_j)$, p_j *exploits* one of its instances a_k^i to fulfill the implementation, where $a_k^i \in AI(a_k)$ the set of instances of a_k. The set of asset instances *exploited* by p_j are noted as $AI(p_j)$. We call *assets* the identified files and *asset instances* their corresponding versions. For example, referring to Table 2, we can identify 1 instance for asset *match.jsp* and 3 instances for asset *style.css*.

We designate *correlations* as the mappings between features and assets, and between features and asset instances. 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 that supposedly contribute in its implementation. Hence, a feature might be correlated

[1] The implementation files of the PVs of the running example are available on: https://github.com/eddyghabachi3s/SoccerManager

Table 1. Product variants with their corresponding features

Product	Feature			
	ManageMatches	AddMatches	ModifyMatches	DeleteMatches
p_1	✓	✓	✓	
p_2	✓	✓	✓	✓
p_3	✓	✓		

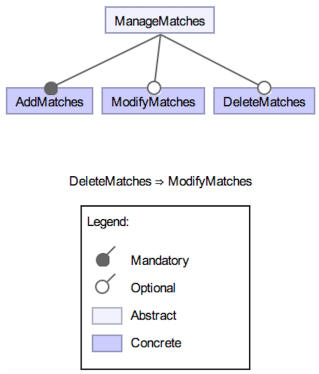

Fig. 1. Running example SPL FM

Table 2. Product variants with an excerpt of their corresponding assets

Product	Asset$^{\text{version}}$
p_1	match.jsp^1 SaveMatch.java^1 style.css^1
p_2	match.jsp^1 SaveMatch.java^1 style.css^2 DeleteMatch.java^1
p_3	match.jsp^1 SaveMatch.java^2 style.css^3

to several assets, and an asset might be correlated to several features as well. We consider that a correlation has to be identified between a feature and an asset instance, if we find at least a product implementing the feature and exploiting the asset instance, with a constraint that no other product exploits the same asset instance without implementing the feature, or implements the feature without exploiting any instance of the asset. Thus, given an instance a^{i} of an asset a, a correlation between a feature f and a^{i} noted as $c(f, a^{\text{i}})$ holds if $\exists\, p_j, f \in F(p_j), a^{\text{i}} \in AI(p_j) \wedge \nexists\, p_k, (a^{\text{i}} \in AI(p_k), f \notin F(p_k) \vee f \in F(p_k), a \notin A(p_k))$. For example, the correlation $c(ModifyMatches, match.jsp^1)$ does not hold, because the same instance $match.jsp^1$ exploited by p_1 and p_2 which implement $ModifyMatches$, is also exploited by p_3 that does not implement $ModifyMatches$. Moreover, the correlation $c(AddMatches, DeleteMatch.java^1)$ does not hold, because except p_2, the products p_1 and p_3 implement the feature $AddMatches$ without exploiting any instance of the asset $DeleteMatch.java$. A correlation between a feature and an asset is identified if there exists at least one of the instances of the asset in correlation with the feature. Thus, given a feature f and an asset a, a correlation $c(f, a)$ holds if $\exists\, a^{\text{i}} \in AI(a) \wedge c(f, a^{\text{i}})$.

We consider an $\mathcal{SPL}$ *complete* if each of its features, assets and asset instances has at least one correlation. To guarantee the *completeness* of an $\mathcal{SPL}$, we impose the following rules: given two products $(p_j, p_k) \in \mathcal{P}$, ① there has to be at least a feature in common between them which is the root feature, ② no two products have exactly the same implementation (same set of asset instances), thus, if $A(p_j) = A(p_k) \Rightarrow AI(p_j) \neq AI(p_k)$, ③ if p_k implements all the features implemented by p_j and more, p_k has to employ all the assets employed by p_j and – not necessarily but most likely – more, thus, if $F(p_j) \subset F(p_k) \Rightarrow A(p_j) \subseteq A(p_k)$.

2.2 Product configuration and derivation

A *restrictive FM* allows an automated derivation of the exact set of products $\mathcal{P}$ provided by the $\mathcal{SPL}$. However, a software engineer might be interested in creating a product that is not provided by the $\mathcal{SPL}$, such as p_4 that implements the features $ManageMatches$, $AddMatches$ and $DeleteMatches$. Thus, we define a *free FM*, a constraint-free version of the *restrictive FM* where all features except the root are optional. Hence, a *free FM* allows a software engineer to select the features required for a *desired* product, in addition to extension points allowing to add new features that are not provided yet by the $\mathcal{SPL}$.

We define a configuration cf consisting of two sets of features: $EF(cf)$ the features required for the desired product and offered by the $\mathcal{SPL}$, and $NF(cf)$ the features required and not offered by the $\mathcal{SPL}$. For instance, the configuration cf_4 relative to the derivation of p_4 has $EF(cf_4) = \{ManageMatches, AddMatches, DeleteMatches\}$ and $NF(cf_4) = \{\phi\}$. When $NF \neq \{\phi\}$, software engineers are asked to determine the assets to be added and/or modified to introduce the new features. Inspired from [13], we categorize the potential products to achieve a certain configuration into three categories: given a product p and a configuration cf, ① p *realizes* cf if $EF(cf) = F(p)$, ② p *covers* cf if $EF(cf) \subset F(p)$, ③ p *contributes in* cf if $EF(cf) \not\subset F(p) \wedge \exists f \mid f \in EF(cf) \wedge f \in F(p)$. If no product *realizes* a certain configuration, several scenarios might be possible to achieve the configuration. Hence, we identify for each configuration cf a set of *configuration scenarios* $\{cs_1(cf), ..., cs_n(cf)\}$. A configuration scenario $cs_i(cf)$ is defined 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 where each p_k *covers* or *contributes in* cf, while $\{f_q, ..., f_s\}$ refers to the *unrequired features* of p_k and $\{f_x, ..., f_z\}$ is $NF(cf)$, if any. Table 3 shows the possible configuration scenarios for cf_4.

A configuration scenario is a way to identify the suitable assets and operations to perform over their instances to achieve the configuration. An asset a is required for a configuration cf if $F(a) \bigcap EF(cf) \neq \{\phi\}$ where $F(a) = \{f_j \mid c(f_j, a)\}$. We define three types of actions that can be performed on an asset instance:

1. Clone and Retain (CRT): consists of cloning a required asset instance and retaining it as it is, without modifying its implementation.
2. Clone and Remove (CRM): consists of cloning a required asset instance, and removing 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): consists of extracting from an asset instance the implementation fragments of some features required by the configuration, and adding them to a cloned asset instance under construction. An ETA action is used only as a subsequent to a CRT or CRM action.

We define an *action* ac as a triple $\langle type, a^{\mathrm{i}}, \{f_j, ..., f_n\}\rangle$, where *type* is one of the action types defined earlier $\{CRT, CRM, ETA\}$. For CRT and CRM,

Table 3. Possible configuration scenarios for configuration cf_4

cs_1	$\langle\{\langle p_2, \{ModifyMatches\}\rangle\}\rangle, \{\phi\}$
cs_2	$\langle\{\langle p_1, \{ModifyMatches\}\rangle, \langle p_2, \{ModifyMatches\}\rangle\}\rangle, \{\phi\}$
cs_3	$\langle\{\langle p_2, \{ModifyMatches\}\rangle, \langle p_3, \{\phi\}\rangle\}\rangle, \{\phi\}$
cs_4	$\langle\{\langle p_1, \{ModifyMatches\}\rangle, \langle p_2, \{ModifyMatches\}\rangle, \langle p_3, \{\phi\}\rangle\}\rangle, \{\phi\}$

a^{i} is the asset instance to clone, while for ETA, a^{i} is the asset instance to extract from. $\{f_j, ..., f_n\}$ is the set of features to be removed or extracted from a^{i}, if *type* is CRM or ETA respectively. Hence, we define an *operation op* as a triple $\langle a, \{ac_1, ..., ac_n\}, a^{\mathrm{i}}\rangle$ where a is the operation asset, $\{ac_1, ..., ac_n\}$ noted as $AC(op)$ is the set of actions to be made to obtain the suitable asset instance a^{i}. For instance, 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 $style.css^4$. Several operations might be identified for a required asset, where only one of them has to be chosen.

A software engineer might be interested in choosing the configuration scenario dealing with the products that she is most familiar with, or that involves the least number of products (i.e. cs_1 involves only p_2), or the one having the least number of operations that require a modification of assets (i.e. cs_3 requires less modifications than cs_1). For these purposes, we auto-generate an FM (see Fig. 2) based on the identified configuration scenarios and operations. The generated FM uses a classic FM formalism, but serves only in supporting the selection of the operations. If one of the operations has a CRT action, it is chosen by default since it does not involve any modification to the asset instance in concern. The generated FM can be configured from several dimensions. A software engineer can make her choice of operations within or outside a configuration scenario, and she can deselect the products or asset instances that she is not familiar with as well, in order to reduce the possible choices.

It is essential to permit the reuse of the newly derived products. Therefore, to enable an incremental evolution of the $\mathcal{SPL}$, we enrich the $\mathcal{SPL}$ with the derived products. We perform a *FAMILIAR merge* operation [14] on the *restrictive FM* and the newly derived product FM to obtain an updated *restrictive FM*. As well, we re-generate the *free FM* and we update the correlations.

3 Experiments and Limitations

We demonstrate the feasibility of our approach on a case study of 8 PVs, by performing an incremental derivation and integration of 5 PVs into an SPL composed initially of 3 PVs. The SPL comprises when it has its 8 PVs a total of 93 features, 271 assets and 296 asset instances with an average of 66 features, 214 assets and 4.7KLOCs per PV. Table 4 illustrates some significant metrics

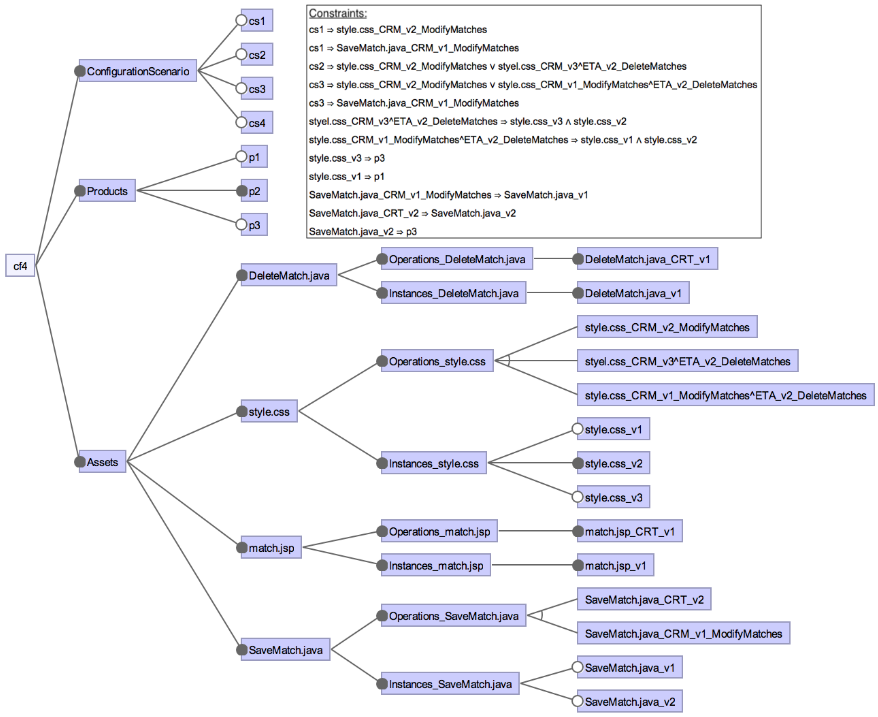

Fig. 2. Generated FM upon configuration of cf_4

that we collected from the configurations. Metrics show that the number of configuration scenarios per configuration increases as long as the SPL becomes rich. Further, despite that the number of required assets can be large, the number of operations to perform might be few. The interest in our approach is that it can identify the assets to be modified and the ones to retain without modification. Further, the deselection of some undesired products or asset instances from the generated FM considerably reduces the choice of a configuration scenario. Moreover, metrics show that, if selection is made by operations regardless the configuration scenarios that they belong to, the number of operations to perform is less, compared to a selection made by configuration scenario. Thus, a software engineer who is familiar with the SPL can rely on this dimension.

A limitation of our approach is that it is dependent on the architecture of the developed SPL. A change in structure or naming of the SPL artifacts affects the identified correlations. However, adhering to the proposed operations during product derivation avoids such inconsistencies. Another limitation is that correlations are identified at file level, while several related works when performing feature location, map features to implementation blocks of several files. Such techniques can be complementary to our approach, since we consider that guidance is the most meaningful when provided at file level.

Table 4. Metrics of 5 sequential configurations to derive new PVs

Configuration	cf_4	cf_5	cf_6	cf_7	cf_8
NB of CSs	7	10	16	48	56
AVG NB of Products per CS	1.714	2.5	3	3.3	4.214
NB of required Assets	185	221	211	211	244
NB of Assets to modify if selection made by OPs	4	3	2	2	2
AVG NB of Assets to modify if selection made by CSs	7.143	3.5	4.5	3.167	2.286
NB of features added by the configuration	0	0	25	0	0
NB of Assets added after derivation	0	0	8	0	24
NB of Asset instances added after derivation	3	1	11	1	25

NB: number — AVG: average — CS: configuration scenario — OP: operation

4 Related Work

Fischer et al. developed the *ECCO* approach [7] that allows an automated derivation of existing PVs and supports the derivation of new PVs by an automated extraction of the required artifacts, and a guidance during the manual completion of the PV. Further, it allows an incremental enrichment of the new PVs. In our approach, we focus on guiding developers in manual derivation, since we consider automated derivation can degrade ownership level and trust of developers in the newly derived products. Martinez et al. proposed a bottom-up extractive approach that migrates PVs from several artifact types into an SPL [11]. The approach performs feature identification when features are not provided, and feature location when features are known. Moreover, it provides word cloud visualization, which helps software engineers to name the identified features. This proposed approach allows an automated derivation of existing and new PVs as well, however, contrarely to our approach the new PVs cannot be incrementally integrated in the SPL. Rubin and Chechik proposed a framework to manage PVs developed using C&O approach [15]. They consider features as the main unit of reuse and they define a set of useful operators to manage PVs and derive new ones. Narwane et al. define operators to investigate traceability between features and assets [13]. Although the functionality of some operators from [15] and [13] are provided by our approach, we consider that integrating these operators in our approach can be an added value.

5 Conclusion and Future Work

In this paper, we presented our approach in guiding software engineers to derive new PVs based on C&O and incrementally integrating them into an SPL. Our experiments showed that the configuration scenarios and operations to perform that we propose upon a new configuration can guide software engineers to construct new PVs, so they can maintain their ownership and trust on the developed PVs since they built it by themselves. As future work, we plan to enhance the provided guidance by a cost estimation for the identified operations, so software engineers can rely on it as an additional parameter during derivation. Further, we aim to compare our approach to related works, and measure its effectiveness in terms of efforts and time saving when compared to the classic C&O approach.

References

1. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Boston (2001)
2. Berger, T., Lettner, D., Rubin, J., Grünbacher, P., Silva, A., Becker, M., Chechik, M., Czarnecki, K.: What is a Feature?: A Qualitative Study of Features in Industrial Software Product Lines. Proceedings of the 19th International Software Product Line Conference. pp.16–25 (2015)
3. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S. : Feature-oriented domain analysis (FODA) feasibility study. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst. (1990)
4. Bagheri. E., Ensan, F., Gasevic, D., Boskovic, M.: Modular feature models: Representation and configuration. Journal of Research and Practice in Information Technology, 43(2):109 (2011)
5. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer Science & Business Media, (2005)
6. Lapeña, R., Ballarin, M., Cetina, C.: Towards Clone-and-own Support: Locating Relevant Methods in Legacy Products. Proceedings of the 20th International Systems and Software Product Line Conference. pp. 194–203. Beijing, China (2016)
7. Fischer, S., Linsbauer L., Lopez-Herrejon, R.E., Egyed, A.: Enhancing clone-and-own with systematic reuse for developing software variants. Proceedings of IEEE International Conference on Software Maintenance and Evolution. pp. 391–400 (2014)
8. Dubinsky, Y., Rubin, J., Berger, T., Duszynski, S., Becker, M., Czarnecki, K.: An exploratory study of cloning in industrial software product lines. 17th European Conference on Software Maintenance and Reengineering (CSMR), pp. 25–34 (2013)
9. Ziadi, T., Henard, C., Papadakis, M., Ziane, M., Le Traon, Y.: Towards a language-independent approach for reverse-engineering of software product lines. Proceedings of the 29th Annual ACM Symposium on Applied Computing. pp. 1064–1071 (2014)
10. Krueger, C.W., van der Linden, F.: Easing the transition to software mass customization. International Workshop on Software Product-Family Engineering, pp. 282–293. Springer (2002)
11. Martinez, J., Ziadi, T., Bissyandé, T.F., Klein, J., Le Traon, Y.: Bottom-up adoption of software product lines: a generic and extensible approach. Proceedings of the 19th International Software Product Line Conference. pp. 101–110 (2015)
12. Assunção, W. K. G., Lopez-Herrejon, R. E. , Linsbauer, L., Vergilio, S. R., Egyed A.: Reengineering legacy applications into software product lines: a systematic mapping. Empirical Software Engineering 22(6): 2972-3016 (2017)
13. Khandu Narwane, G., Galindo Duarte, J., Narayanan Krishna, S. and Benavides, D., and Millo, J., Ramesh, S.: Traceability analyses between features and assets in software product lines. Entropy MDPI (2016)
14. Acher, M., Collet, P., Lahire, P., France, R.B: Familiar: A domain-specific language for large scale management of feature models. In Science of Computer Programming, Vol. 78. Elsevier, pp. 657–681 (2013)
15. Rubin, J., Czarnecki, K., Chechik, M.: Managing cloned variants: a framework and experience. Proceedings of the 17th International Software Product Line Conference. pp. 101–110 (2013)