Safe evolution of product populations and multi product lines

Leopoldo Teixeira
Federal University of Pernambuco
lmt@cin.ufpe.br

Paulo Borba
Federal University of Pernambuco
phmb@cin.ufpe.br

Rohit Gheyi
Federal University of Campina Grande
rohit@dsc.ufcg.edu.br

ABSTRACT

A product line is often developed in the context of a set of related product lines. When supporting separate feature development, we might have product populations, with product line versions being simultaneously developed in different branches. Multi product lines involve a number of product lines that depend on each other. A product line refinement notion formalizes safe evolution, but this is not sufficient for reasoning over sets of product lines. We propose refinement notions and compositionality properties that help to explain how we can support modular development in these contexts. Thus, we formally define the foundations for safe and modular evolution of product populations and multi product lines, enabling developers to perform changes in a systematic manner.

CCS Concepts

•Software and its engineering → Software product lines; Software evolution;

1. INTRODUCTION

Due to technical and market reasons, a Software Product Line[1] is often developed in the context of a set of related product lines [17, 5]. In the context of product populations [17] we have product lines sharing assets and covering domains with overlapping functionality, maybe even being simultaneously developed in different branches and repositories, when supporting separate feature development [5]. Multi product lines combine and compose product lines, which might be independently developed, but depend on each other to generate a final product [8, 9, 13]. When evolving such sets of product lines, sometimes it is important to check if changes preserve the observable behavior of existing products.

A theory of product line refinement [4] formalizes safe evolution as a refinement notion, providing guidance when behavior should be preserved after changes, when we want to make sure that changes do not impact existing products. It takes the broader view of refinement [6][2] as a relation that preserves behavior to assure safe evolution. The notion considers variability and configuration models, and is compositional with respect to changes in such elements [4].

While we can rely on this theory for reasoning about individual product line evolution, it is not sufficient for reasoning over sets of product lines [17, 8, 9, 13]. For modularity's sake, safe evolution of an individual member should not affect the related product lines in the same set. Neglecting such compositionality issues can lead to bugs. For example, bug #8557[3] shows that a feature change in the Linux kernel causes Ubuntu 4.10 to not work at all in some systems. Similar scenarios are found in the bug tracking systems of other Linux distributions, that resemble multi product lines.

To understand the compositionality issues and establish the foundations for safe and modular evolution in these contexts, we propose refinement notions for product populations and multi product lines, which can provide guidelines that could minimize adoption and maintenance costs. For product populations, it enables reasoning over scenarios such as merging product line versions developed in different branches or repositories, to provide guarantees that there were no unintended interactions in the evolution of the different versions. For multi product lines, it supports scenarios such as creating a multi product line out of an existing large product line, which can be useful for enabling distributed development [5].

More importantly, we also specify and compositionality theorems, which reveal conditions that guarantee safe evolution and help to explain how we can support modular development. We provide additional evidence that changes to individual product lines of a multi product line must be harmonized [8]. The identified conditions can also be useful for the implementation of refinement verification tools. To specify and prove our theories and compositionality properties, we use the Prototype Verification System (PVS). Specification and proof files are available at our online appendix.[4]

We organize the remainder of the text as follows. In Section 2 we overview product populations and multi product lines. We present our formalization, refinement notions, and compositionality properties in Section 3. We discuss related work in Section 4, and conclude in Section 5.

2. OVERVIEW

A product population is a set of product families [17], useful when we need to deal with a wide scope of products, reusing functionality across related domains. We observe this in companies and open source projects that adopt distributed development, whether by discipline, along a software supply chain, or by accident [5]. For

[1] Hereafter, product line.

[2] Feature-oriented programming [10] uses this term in the quite different sense of overriding or adding extra behavior to assets.

[3] https://bugs.launchpad.net/bugs/8557

[4] http://www.leopoldomt.com/papers/mpl-populations

SPLC 2015, July 20 - 24, 2015, Nashville, TN, USA

ISBN 978-1-4503-3613-0/15/07...$15.00

DOI: http://dx.doi.org/10.1145/2791060.2791084

example, TaRGeT,[5] a product line of model-based test generation tools. In a give moment of time, between the circles in Figure 1, another organization became interested in using the tool. Since some of the features were not open source, a separate branch was set up. In this branch, the new organization could use the open source features and develop new ones, according to its requirements.

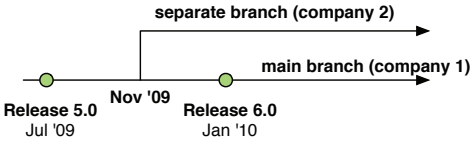

Figure 1: TaRGeT evolution

We also observe this branching model in other projects. In TaRGeT, besides development reasons, there was also the issue of features that could not be shared among organizations. Nonetheless, some features developed in the separate branch were later integrated to the main one, so there was a need to to relate product lines through merging. This way, it would be possible to ensure that users could still use existing products without observable behavior changes after the merge. Other scenarios can also benefit from relating sets of product lines, such as splitting a large and complex product line into smaller ones due to business reasons.

Multi product lines [8, 13] combine and compose multiple product lines, that are independently developed but depend on each other to generate a product. For example, consider the Ubuntu Linux distribution, which integrates a number of configurable systems, including the Linux kernel, and needs to manage dependencies among these systems to generate products [8]. A previous study shows that in the Debian distribution, although packages are developed in separate, they are subject to complex dependencies [7].

Figure 2 shows a simplified Ubuntu multi product line using feature models. **LK** stands for the Linux kernel, where each product targets an architecture (*Arch*). *PCI_MSI* is a network driver, and *Default* is the default kernel configuration used by the distribution. **BB** stands for BusyBox, which combines common UNIX utilities into a single small executable. *GETOPT_LONG* enables parsing long command-line options. Finally, **KI** stands for Kickstart, an automated installation method, that can optionally store configuration files on a server. Dependencies among the product lines happen, such as *Kickstart* requires *GETOPT_LONG*. Dots and open ended lines denote irrelevant features for this example.

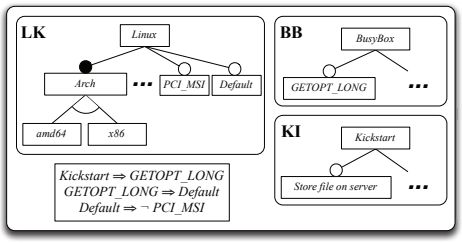

Figure 2: Simplified Ubuntu multi product line.

Dependencies must be taken into account when configuring multi product lines [8, 13]. A product actually consists of combining products from the product lines that satisfy such dependencies. So, using pairs of product lines and their respective configurations, although each pair in {(**BB**,{*BusyBox*, *GETOPT_LONG*,...}), (**LK**, {*Linux*,*Arch*,*amd64*,...}), (**KI**, {*Kickstart*, ...})} individually consists of valid configurations from the constituent product lines, their composition violates the constraint *GETOPT_LONG* ⇒ *Default*.

To enable modular development, it is important to ensure that, when maintaining an individual product line, we preserve the behavior of existing products. The Ubuntu bug #293586[6] shows that after an evolution, *GETOPT_LONG* was no longer associated to the default configuration. This affected almost all uses of Kickstart. Another example is the previously mentioned bug #8557, where a feature was enabled by default after a change. This illustrates the need for relating sets of product lines to guarantee safe evolution.

3. FORMALIZATION

We extend the product line refinement theory [4], briefly introduced in Section 3.1. The following sections specify refinement notions and compositionality properties for product populations and multi product lines, using a simplified PVS notation.

3.1 Product Line Refinement Theory

The refinement theory defines a product line as three elements that jointly generate well-formed products: (i) a variability model F; (ii) an asset mapping A, providing the means for referring to assets; (iii) a configuration knowledge K, mapping features to assets. It does not rely on particular languages that can be used to represent such elements, as long as some assumptions are satisfied. For example, for a variability model F, it only assumes a generic function, denoted by $[\![F]\!]$, that defines its semantics as a set of configurations. Therefore, assuming a semantics function for generating products according to a configuration c, represented as $[\![K]\!]^A_c$, we formalize a product line as $\forall c \in [\![F]\!] \cdot wf([\![K]\!]^A_c)$. We use $[\![L]\!]$ to denote the set of products of a product line, given by $\{[\![K]\!]^A_c \mid c \in [\![F]\!]\}$.

Assuming a relation $\sqsubseteq$ that establishes refinement of an asset set, the product line refinement notion lifts behavior preservation from products to product lines, overloading the symbol. We use $L \sqsubseteq L'$ to denote that the product line L is refined by L', whenever each product p generated by L is refined by some product p' of L', formalized as $\forall p \in [\![L]\!] \cdot \exists p' \in [\![L']\!] \cdot p \sqsubseteq p'$. The definition focuses on the generated products, therefore, feature names do not matter. Refinements are safe transformations in the sense that we can change a product line without impacting existing users, because behavior of the existing products is preserved. For example, by improving the internal structure of a feature implementation. Certain kinds of useful changes are not refinements, such as bug fixes, where the intention is to change behavior. Nonetheless, studies about the evolution of the Linux kernel [12] show that most of the changes consist of adding new features and modifying existing ones. Product line refinement is compositional, so F, K, and A can be independently modified under certain conditions [4]. However, the existing theory does not handle relating multiple product lines, as needed to deal with safe evolution scenarios involving sets of product lines.

3.2 Product populations

Assuming that a set of product lines defines a product population, we specify the corresponding type *Population* as a set of product lines (*PL*), represented as $\mathscr{P}[PL]$. We also use $[\![P]\!]$ to denote the set of products of P, given by $\bigcup_{L \in P} [\![L]\!]$. Product population refinement then follows the same reasoning as product line refinement, preserve behavior of existing products in the original population. We again lift the notion of behavior preservation, this time from product lines to product populations.

[5] http://www.cin.ufpe.br/~target/

[6] https://bugs.launchpad.net/bugs/293586

DEFINITION 1. *A product population P is refined by another P', denoted by $P \sqsubseteq P'$ whenever $\forall p \in [\![P]\!] \cdot \exists p' \in [\![P']\!] \cdot p \sqsubseteq p'$*

We define refinement in terms of the actual products generated by the population. Defining it using the product line refinement notion ($\forall L \in P \cdot \exists L' \in P' \cdot L \sqsubseteq L'$) would restrict the changes that we could support. For example, when splitting a product line, this alternative definition would not apply, while Definition 1 supports it. We also prove that this notion is a pre-order. For conciseness, we omit some theorems from the text, but they are available in the online appendix.

We then establish conditions for evolving product lines that belong to a population in an independent and parallel way. To prove this, we use an auxiliary lemma establishing that product population semantics is distributive, that is, for a product line L and product population P, we have that $[\![L \cup P]\!] = [\![L]\!] \cup [\![P]\!]$. We use $\cup$ both to denote set union and insertion of an element to a set — in the case of $e \cup s$, for an element e and set s, as an abbreviation for $\{e\} \cup s$. In the following, we then establish that safely evolving a product line that is part of a population implies safe evolution of the entire population. This theorem also enables us to reuse the results from safe product line evolution [11, 2, 4].

THEOREM 1. *For product lines L and L', and product population P, if $L \sqsubseteq L'$ then $L \cup P \sqsubseteq L' \cup P$.*

Proof: *Assume that $L \sqsubseteq L'$. By Definition 1, we have to prove, for an arbitrary $p \in [\![L \cup P]\!]$ that $\exists p' \in [\![L' \cup P]\!] \cdot p \sqsubseteq p'$. For an arbitrary p we have that $p \in [\![L]\!] \cup [\![P]\!]$ by the auxiliary lemma. By case analysis, first consider that $p \in [\![L]\!]$. The proof follows from our assumption, since for all $p \in [\![L]\!]$, there is a $p' \in [\![L']\!]$ such that $p \sqsubseteq p'$. Let us now consider the case where $p \in [\![P]\!]$. The proof follows by letting p' be p and from asset set refinement reflexivity [4].*

Compositionality theorems provide a framework for reasoning about modularity. For product populations, since products are independently generated, we only need to ensure refinement of the individual product lines to guarantee safe evolution of the product population. This reflects what happens in distributed development, where branches evolve independently, as illustrated in Figure 1. Each branch corresponds to a different product line, due to different requirements and even legal issues, and some features developed in a separate branch were later integrated to the main branch as optional features. The refinement notion is useful to relate sets of product lines, ensuring that integration does not change the observable behavior of existing products. Another application is the development of product variants using clone-and-own [14, 15, 3]. To obtain the benefits associated to product lines with reduced upfront investment, we can bootstrap a product line from existing products [14], using product population refinement to ensure that we preserve behavior of existing products while doing so.

3.3 Multi Product Lines

Multi product lines also consist of sets of product lines, with constraints specifying how product lines relate to each other. Therefore, the products are built by composing subproducts generated from the product lines in the set, taking into account the constraints. We represent configurations as a set of pairs (l,c) formed by a product line $l = (F,A,K)$ and one of its product configurations $c \in [\![F]\!]$, which make the type PC. For a pair (l,c) we use $[\![(l,c)]\!]$ to abstract the notation $[\![K]\!]^A_c$. We use $[\![\_]\!](S : \mathscr{F}[PC])$ to denote that for a finite set of such pairs, the resulting product is the union of the asset sets generated by each pair, given by $\bigcup_{(l,c)\in S} [\![(l,c)]\!]$. In Figure 2, this corresponds to combining the assets of the subproducts from the **LK**, **BB**, and **KI** product lines.

We formalize multi product lines as a finite set of product lines and a finite set of constraints, as Figure 2 illustrates. Following the same reasoning for the product line elements [4], we abstract from particular languages, so we just assume *Constraint* as an uninterpreted type, which imposes no assumptions over the defined type. Likewise, we do not specify how a multi product line defines its valid configurations. Given a set of product lines S and a set of constraints C relating them, we use $confs(S,C)$ to denote the function that yields the set of product configurations that obey the constraints. We then formalize multi product lines similarly as we do for product lines. For a set of product lines S and a set of constraints C, we say that (S,C) is a multi product line (*MPL*) when all of its products are well-formed: $\forall pcs \in confs(S,C) \cdot wf([\![pcs]\!])$.

For generality, we do not precisely specify what *confs* does. Nevertheless, we use axioms, as an interface to provide guidance and establish how this function should be specified. First, we require that all valid configurations from a multi product line m only contain pairs in which the product line is a member of m, with at most one pair for each constituent product line (**Axiom 1**). This prevents *confs* from generating sets with more than one configuration from the same product line. If needed, we can use cardinality-based variability modeling to simulate having two configurations from the same product line. We also do not demand that products must have configurations from all product lines in the multi product line. This enables the situation where we have an optional product line, that is, some products from the multi product line do not have any functionality coming from it. For example, in Figure 2, this allows us to generate a Ubuntu distribution without the *Kickstart* functionality. That is, without any feature from the **KI** product line.

We also require *confs* to preserve the set of configurations when adding a new member to the multi product line without changing the cross-product line constraints (**Axiom 2**). This is possible since we establish that the valid configurations from a multi product line do not need to contain configurations from all product lines. Reflecting what happens in practice, this allows adding new software packages (product lines) with additional functionality to Ubuntu (see Figure 2), such as extra filesystem modules, for instance.

Finally, as we consider sets of product lines, we need to take into account that we might remove a product line from the set. Therefore, we establish that if we remove the product line L from a multi product line, *confs* must preserve the set of configurations that do not receive any contribution from L (**Axiom 3**). That is, after the change, we should still be able to generate any product that has no functionality from L. So, if we remove the **KI** product line from the Ubuntu example in Figure 2, this means that *confs* must continue generating all product configurations without the *Kickstart* functionality. This way, we avoid impacting existing users.

We then formalize the semantics of a multi product line, that is, the function that yields the actual set of products, by combining the subproducts for each product configuration that *confs* yields. In our formalization, we uniformly represent multi product line products and product line subproducts as finite asset sets. So, for a multi product line (S,C), we use $[\![(S,C)]\!]$ to denote its set of products, given by $\{\ p : \mathscr{F}[Asset] \mid \exists\ pcs \in confs(S,C) \cdot p = [\![pcs]\!]\ \}$

Multi product line refinement then follows the same structure of both product line and product population refinement. We state that each product in the original multi product line must have a correspondent product in the refined one.

DEFINITION 2. *A multi product line (S,C) is refined by another multi product line (S',C'), denoted by $(S,C) \sqsubseteq (S',C')$, whenever $\forall p \in [\![(S,C)]\!] \cdot \exists p' \in [\![(S',C')]\!] \cdot p \sqsubseteq p'$*

The difference from the previously discussed refinement notions is the semantics function used to generate products, which works composing subproducts from different product lines. We also prove that multi product line refinement is a pre-order. Similar to what we

discuss for product populations, defining multi product line refinement in terms of product line refinement is also not sufficient. The constraints must be taken into account. For example, as product line refinement does not care about feature names, a useful refinement, such as feature renaming, could break existing products of the multi product line, as this possibly affects the constraints.

We are also interested on properties that enable reasoning about safe and modular evolution of the product lines that are part of a multi product line. However, the natural dependencies among the members of a multi product line, which we express as sets of constraints, can limit compositionality. As illustrated in Figure 2, constraints might relate features of different product lines. So, different from compositionality in the product population context, safe evolution of a product line that is part of a multi product line does not necessarily implies on safe evolution of the multi product line.

As discussed, a mere feature renaming in a product line could break existing products, if we do not update the constraints. Thus, there are different ways in which we can evolve multi product lines, leading to different conditions that must be checked to ensure refinement. Therefore, we can specify a number of compositionality theorems, revealing specific conditions for modularly evolving a multi product line in different situations, which include refining a product line without changing the constraints, removing and merging product lines, and refining a product line together with the constraints. Here we present a particular theorem and its proof, but we informally discuss other situations in what follows.

A particular scenario is when we modularly refine a constituent product line without changing the constraints. That is, we safely evolve a product line maintaining all existing configurations, due to a refactoring, or by adding an optional feature, besides other scenarios. The product line refinement notion is insensitive to feature names. So, in this case we require a stronger refinement notion, sensitive to the name and semantics associated to each feature. It states that all configurations from the variability model of a product line L are also present in the resulting product line L', and the correspondent product is a refinement. We need to preserve configurations since we must ensure that we do not break the constraints, as it happened on the Ubuntu bug #293586. We use $(F,A,K) \preceq (F',A',K')$ to denote $\forall c \in [\![F]\!] \cdot c \in [\![F']\!] \wedge [\![K]\!]^A_c \sqsubseteq [\![K']\!]^{A'}_c$. For each c in the original product line, the same c generates a refined product in the resulting product line.

THEOREM 2. *Modularly evolving a constituent product line: For product lines L and L', a finite set of product lines S not containing L or L', and a finite set of constraints C, if $L \preceq L'$ and $wf(L' \cup S, C)$, then $(L \cup S, C) \sqsubseteq (L' \cup S, C)$.*

Proof: *Assume that $L \preceq L'$ and $wf(L' \cup S, C)$. By Definition 2, we have to prove, for an arbitrary $p \in [\![(L \cup S, C)]\!]$, that $\exists p' \in [\![(L' \cup S, C)]\!] \cdot p \sqsubseteq p'$. For such an arbitrary p, we have that $\exists\, pcs \in confs(L \cup S, C) \cdot p = [\![pcs]\!]$. Let ps be such pcs. By case analysis, first consider that $L \notin ps$. By Axiom 3, we have that $pcs \in confs(S, C)$. By Axiom 2, we then have that $pcs \in confs(L' \cup S, C)$. The proof follows from asset set refinement reflexivity [4]. Let us now consider the case where $L \in ps$, which means that there is one pair (L, c) in ps such that c is a configuration from L. From $L \preceq L'$, we have that all configurations from L are in L' and refine existing products. Therefore, we replace (L, c) with (L', c) in ps and the proof follows from the compositionality of asset set refinement [4].*

We do not preclude the new product line from having more products than the original. However, when $[\![F]\!]$ is equivalent to $[\![F']\!]$, the refined product line generates the exact same configurations as the original. In this case, we do not need to require well-formedness as a condition for the theorem, since we can prove it by compositionality of the asset set refinement notion [4]. Also, we require that the refined product line L' is not an element of S. This forbids the pathological situation where we would transform the product line L into a product line L' which already exists in the multi product line, which is one of the particular safe evolution scenarios we can also address with different compositionality theorems. A general scenario would be evolving a product line together with the constraints. Then, we need to impose strong restrictions for specifying the compositionality theorem, which reflects results from an expert survey, where participants pointed out that changes to individual product lines must be harmonized [8]. Another scenario would be evolving only the constraints. Then, we need to ensure that the existing configurations generated by *confs* are still generated.

Adding an optional feature to the BusyBox (**BB**) product line in Figure 2, without changing the constraints, is an example of modularly evolving a constituent product line, as established in Theorem 2. After the change, the new product line still generates all of the original configurations, besides the ones containing the new feature, which is compliant with the necessary condition for the theorem. Through reflexivity, this results in safe evolution of the multi product line, as established in Definition 2. Asset refinement without changes to the variability model fits in the particular case of modularly evolving a constituent product line where we generate the exact same configurations. We only need to check that the particular asset set was refined, avoiding the need to check all products. The theorem conditions are helpful to reveal conditions for modularly evolving product lines and still assure safe evolution of the multi product line, and are also useful for implementing tools that could avoid problems such as the Ubuntu bugs.

As discussed, not all evolution scenarios are refinements, such as feature retirement. Nonetheless, the refinement theories indeed support some scenarios of feature retirement. For example, when the retired feature is merged with an existing feature. This happens when developers add the capabilities of old device drivers into recent ones, consequently removing the deprecated drivers [12]. When removing a product line L from the multi product line, we need to make sure that each product that has some contribution from L has a corresponding, behavior preserving, product in the refined multi product line. In summary, we must be careful when changing feature names, constraints, or eliminating existing configurations. In fact, these configurations and the associated feature names might be referred by the constraints. This can preclude safe and modular multi product line evolution, since we might change the observable behavior of existing products and thus impact users.

One could argue that we can model a multi product line as a large product line with its associated artifacts. However, organizations adopt multi product lines because they are structured by composing products coming from different sources or teams. Additionally, we would lose some of the modularity benefits. By structuring large product lines, we could indeed use the product line refinement theory to reason about evolving variability models and assets in separate. However, we would not be able to support reasoning about changing different parts at the same time, since we would always need to reason about the entire product line. This is possible using the multi product line formalization we present here, since we can reason about individual product lines modularly using the compositionality theorems. The theory also allows us to safely transform a large product line into a multi product line, and vice-versa. We could also formalize product populations in a different way, in terms of multi product lines. To do so, we would not have any constraints among the product lines, and the *confs* function would preclude combining subproducts from the constituent product lines to generate the final product. For simplicity, we choose the more direct way of formalizing populations presented here.

4. RELATED WORK

Rubin and Chechik used the product line refinement theory we rely on for combining products into a product line [14]. They presented an operator and proved its correctness according to the product line refinement notion. The population refinement notion we present is more comprehensive than solely relying on product line refinement. They also proposed a framework for managing related product variants realized via cloning [15], defining abstract operators for expressing maintenance scenarios. Antkiewicz et al. proposed virtual platform [3] as an adoption strategy to ease the transition from cloning to systematic product line engineering approaches. Although these works do not focus on ensuring safe evolution through their operators and strategies, we could leverage the population refinement notion to provide safety when applicable.

Some works focused on modeling, structuring, and analyzing multi product lines, but also approaching evolution [13, 8, 9, 5, 16]. We abstract from implementation and modeling details, and focus on formalizing key concepts and the conditions for safe evolution, aiming to explain when, and in which sense, multi product line development can be done modularly.

Rosenmüller et al. presented an approach to model and configure multi product lines [13], using composition models to combine product lines. They also support automating the configuration process. Their work is complementary, as we could use their models as a concrete instantiation of the multi product lines theory. Moreover, their configurators could be used for implementing the *confs* function. Holl et al. performed a systematic review and an expert survey on multi product line capabilities [8]. "Support for multi product line evolution" was suggested by 30% of the participants. They discuss that evolution is important, as changes to individual product lines must be synchronized. They also proposed an approach to manage dependencies during the configuration process [9], providing tool support. Relating to our work, this approach could be useful for updating the constraints of a multi product line.

Acher proposed FAMILIAR, a language for managing multiple variability models [1], including operators for merging feature models. We can use the language and its operations to implement the *confs* function, as well as reasoning about changes to a feature model to verify conditions needed by compositionality theorems, such as modularly evolving a constituent product line.

We can view ecosystems as multi product lines where the variant space is open. Seidl and Assman presented an ecosystem variability modeling notation [16], capturing temporal information to analyze evolution over time. We go beyond variability models, but do not model temporal information. Brummermann et al. formalized variability composition in information system ecosystems using partial configurations, showing how it helps to address distributed evolution [5]. We abstract implementation details, but could use this approach to the *confs* function and avoid inconsistencies, by checking the conditions from the compositionality theorems.

5. CONCLUSIONS

In this work, we extend the product line refinement theory [4] with refinement notions for product populations and multi product lines, to establish the foundations for safe and modular evolution in these contexts. We also establish and prove compositionality theorems, which establish conditions for individually evolving product lines when reasoning over sets of related product lines. This enables us to reuse and apply the results from product line safe evolution [11, 2, 4], and helps to explain how modular development can be supported in both approaches. We intend to further investigate modularity issues for safe evolution of product populations and multi product lines, understanding their evolution patterns and deriving refinement templates [11, 12].

Acknowledgments

This work was partially supported by the National Institute of Science and Technology for Software Engineering,[7] funded by CNPq (573964/2008-4) and FACEPE (APQ 0388-1.03/14). We acknowledge financial support from FACEPE grant APQ-0570-1.03/14, and CNPq scholarships and grants 477943/2013-6, 306610/2013-2.

6. REFERENCES

[1] M. Acher. *Managing Multiple Feature Models: Foundations, Language, and Applications*. PhD thesis, University of Nice Sophia Antipolis, 2011.

[2] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. Refactoring product lines. In *GPCE*, 2006.

[3] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, S. Stănciulescu, A. Wąsowski, and I. Schaefer. Flexible product line engineering with a virtual platform. In *NIER'2014*, 2014.

[4] P. Borba, L. Teixeira, and R. Gheyi. A theory of software product line refinement. *Theoretical Computer Science*, 455:2 – 30, 2012.

[5] H. Brummermann, M. Keunecke, and K. Schmid. Formalizing distributed evolution of variability in information system ecosystems. In *VaMoS*, 2012.

[6] E. Dijkstra. *Notes on structured programming*. Academic Press, 1971.

[7] J. González-Barahona, G. Robles, M. Michlmayr, J. Amor, and D. Germán. Macro-level software evolution: a case study of a large software compilation. *Empirical Software Engineering*, 14(3):262–285, 2009.

[8] G. Holl, P. Grünbacher, and R. Rabiser. A systematic review and an expert survey on capabilities supporting multi product lines. *Inf. and Software Technology*, 54(8):828–852, 2012.

[9] G. Holl, D. Thaller, P. Grünbacher, and C. Elsner. Managing emerging configuration dependencies in multi product lines. In *VaMoS*, 2012.

[10] J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *ICSE*, 2006.

[11] L. Neves, P. Borba, V. Alves, L. Turnes, L. Teixeira, D. Sena, and U. Kulezsa. Safe evolution templates for software product lines. *J. Syst. Softw.*, Accepted for publication, 2015.

[12] L. Passos, J. Guo, L. Teixeira, K. Czarnecki, A. Wasowski, and P. Borba. Coevolution of variability models and related artifacts: a case study from the Linux kernel. In *SPLC*, 2013.

[13] M. Rosenmüller and N. Siegmund. Automating the configuration of Multi Software Product Lines. In *VaMoS*, 2010.

[14] J. Rubin and M. Chechik. Combining related products into product lines. In *FASE*, 2012.

[15] J. Rubin and M. Chechik. A framework for managing cloned product variants. In *NIER*, 2013.

[16] C. Seidl and U. Assmann. Towards modeling and analyzing variability in evolving software ecosystems. In *VaMoS*, 2013.

[17] R. van Ommering. Building product populations with sofwtare components. In *ICSE*, 2002.

[7] http://www.ines.org.br