Partially Safe Evolution of Software Product Lines

Gabriela Sampaio
gcs@cin.ufpe.br

Paulo Borba
phmb@cin.ufpe.br

Leopoldo Teixeira
lmt@cin.ufpe.br

Informatics Center
Federal University of Pernambuco
Recife, Brazil

ABSTRACT

A key challenge developers might face when evolving a product line is not to inadvertently affect users of existing products. In refactoring and conservative extension scenarios, we can avoid this problem by checking for behavior preservation, either by testing the generated products or by using formal theories. Product line refinement theories support that by requiring behavior preservation for all existing products. However, in many evolution scenarios, such as bug fixing, there is a high chance that only some of the products are refined. To support developers in these and other non full-refinement situations, we define a theory of partial product line refinement that helps to precisely understand which products should not be affected by an evolution scenario. This provides a kind of impact analysis that could, for example, reduce test effort, since products not affected do not need to be tested. Additionally, we formally derive a catalog of eight partial refinement templates that capture evolution scenarios, and associated preconditions, not covered before. Finally, by analyzing 79218 commits from the Linux repository, we find evidence that the proposed templates could cover a number of practical evolution scenarios.

CCS Concepts

•Software and its engineering → Software product lines; Formal methods;

Keywords

product line evolution, product line maintenance, product line refinement

1. INTRODUCTION

When evolving a product line [2, 16], it is often important to make sure that the evolution is safe in the sense that existing users are not inadvertently affected by the performed changes. This safe evolution concept [12] is formalized by a refinement notion [3] that requires every product of the initial product line to have compatible behavior with at least one product of the newly evolved product line.[1] This is useful to support developers in a number of evolution scenarios, helping them to make sure that the changes they make do not have unintended impact. For instance, users might simply need to refactor assets, or even add optional features, and these are guaranteed not to affect existing products, provided that certain conditions are observed. The refinement notion and its associated transformation templates help us to precisely capture those conditions.

Although these notions of product line safe evolution and refinement are useful in many practical evolution scenarios, they are too demanding for other scenarios because they require all products to preserve behavior. Nevertheless, we argue here that we could still support developers even when that does not apply. For example, adding functionality to an asset changes the behavior of all products that use that asset, so this is often not a product line refinement. However, the behavior of products that do not use the modified asset should not be affected. So we could still provide behavior preserving guarantees for a proper subset of the products in a product line.

This kind of partial guarantee can be useful as an impact analysis for developers to be aware of which products are affected in an evolution scenario. They could, for instance, avoid checking behavior preservation of the refined products, focusing only on testing the new functionality on the subset of products that are impacted by the changes. A notion of partially safe product line evolution could assist developers by providing this kind of weaker, but still useful, guarantee that covers common evolution scenarios not supported by refinement. This partially safe evolution concept can be helpful not only in a practical product line development context, but also in formal development situations and in building tools that support product line development.

In fact, many evolution scenarios found in practice do not characterize a full refinement. A bug fix, or changing a top level (child of root) feature from optional to mandatory, for example, are not full refinements because not all original products are refined. More specifically, in the first situation, products containing the files changed due to the bug fix are not refined; the other products, however, have the same behavior since they are not changed. When a feature is transformed from optional to mandatory, products that already had the changed feature are refined because they will

SPLC '16, September 16 - 23, 2016, Beijing, China

ACM ISBN 978-1-4503-4050-2/16/09…$15.00

DOI: http://dx.doi.org/10.1145/2934466.2934482

[1] "Full refinement" denotes the existing refinement notion [3], which requires the full product set to be refined.

present the same behavior in the new product line. However, products that did not have the feature do not preserve behavior because, in the new line, they will present the extra behavior associated to the changed feature. Furthermore, Passos et al. [15] examined commits of the Linux repository history,[2] and found that feature removal, which is not a refinement unless the feature is dead or has no behavior, often occurs. The partially safe evolution notion we propose here can address these cases by requiring refinement for a proper subset of the product line products. Transformation templates derived from this notion capture the context and required conditions for a number of scenarios, and precisely provide the subset of refined products for those cases. For example, in the feature removal scenario, the template could guarantee that products that did not have the removed feature are refined.

We formalize the partially safe evolution notion in terms of a partial refinement notion. As discussed, it only requires behavior preservation for a subset of the existing products in a product line.[3] For scenarios where a change is intended to refine all products, such as changing a feature from mandatory to optional, developers should rather use the full refinement notion. Hence, they might choose to make use of the full and partial refinement notions depending on the situation. Evolution in practice often interleaves different kinds of changes, ranging from full refinement to no refinement scenarios. So, to support practitioners, we derive a number of properties, including that safe and partially safe evolution transformations, when applied in different orders, might lead to the same resulting product line. For example, developers could refine an asset and then remove a feature, or apply these transformations in the opposite order, and still reach the same target. In addition, we propose transformation templates representing abstractions of partial refinement situations encountered in practice. Templates work as a guide for developers. Instead of reasoning over refinement notions, they can use templates by means of pattern matching, which can also be tool supported. The partial evolution templates precisely determine which subset of products is refined for each situation; developers might even obtain this subset automatically. So our templates effectively provide a change impact analysis.

To evaluate the applicability of our templates, we use the FEVER tool [8] to automatically analyze 79218 evolution scenarios of the Linux repository. We found a number of instances of the templates in the commit history and confirm that they could have been applied, thus reinforcing the applicability of our templates. We also formalize the concepts and prove properties and soundness of the templates in the Prototype Verification System (PVS) [14].

To summarize, the main contributions of this work are (i) a new concept of partial product line refinement that covers partially safe evolution scenarios, (ii) a number of properties to support users not only in partially safe evolution scenarios, but also when these transformations are combined with safe evolution ones, (iii) eight patterns that represent partial safe evolution scenarios to guide developers and (iv) evidence of applicability of our templates, based on an analysis of 79218 evolution scenarios of the Linux system.

This paper is organized as follow. In Section 2, we present a motivating example from the Linux repository. We introduce the partial refinement theory in Section 3 and relate it with the full refinement theory. In Section 4, we present a template catalog. We present evaluation results and related work in Section 5 and Section 6, respectively. Finally, we conclude in Section 7.

2. MOTIVATING EXAMPLE

To illustrate a common evolution scenario not covered by the product line refinement notion, we refer to commit *ae3e4c2776*[4] of the Linux repository history. It basically consists of a feature removal scenario. Feature *LEDS_RENESAS_TPU* represents a LED driver in the Linux system. *LEDS_RENESAS_TPU* was removed because it was superseded by the preexisting generic *PWM_RENESAS_TPU* driver. The commit changes are illustrated in Listing 1, 2 and 3. We use the "−" symbol in each line to indicate that it was removed from the file.

In Listing 1, we observe changes to a Linux Kconfig file,[5] which models features and their properties, and plays a similar role to feature models and other variability models. Statements in Kconfig declare features by indicating their names, types (the illustrated one is a *boolean* that can assume y or n, when it is selected or not, respectively) and relations with other features, as specified in Lines 3 and 4. In this case *LEDS_RENESAS_TPU* depends on *LEDS_CLASS*, *HAVE_CLK* and *GPIOLIB*. Thus, the former can only be selected if the three other features are included in the product. In terms of feature models, this condition is akin to establishing *LEDS_RENESAS_TPU* as a descendant of those features.

Listing 1: "drivers/leds/Kconfig"

```
1 -config LEDS_RENESAS_TPU
2 -     bool ''LED support for Renesas TPU''
3 -     depends on LEDS_CLASS=y && HAVE_CLK
4 -             && GPIOLIB
5 -     help
6 -     ...
```

Listing 2: "drivers/leds/Makefile "

```
1 -obj-$(CONFIG_LEDS_RENESAS_TPU) +=
2           leds-renesas-tpu.o
```

Listing 3: "drivers/leds/leds-renesas-tpu.c"

```
1 -#include <linux/module.h>
2 -#include <linux/init.h>
3 - ...
4 -MODULE_LICENSE(''GPL v2'');
```

The *LEDS_RENESAS_TPU* feature is implemented by the *leds-renesas-tpu.o* asset, as we can see in the makefile in Listing 2. These files represent Linux configuration knowledge, relating feature expressions (presence conditions) to

[2]Linux repository is available at http://github.com/torvalds/linux.

[3]We use the "partial refinement" term to denote the new refinement notion, which requires refinement only for a subset of the original products from a product line.

[4]Feature removal commit http://github.com/torvalds/linux/commit/ae3e4c2776. Laurent Pinchart commited on Jul 16, 2013; version v3.12-rc1.

[5]Kconfig language documentation https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt.

asset names. This mapping was removed, since the intention was to remove the feature. However, a feature is only completely removed when its implementation is deleted as well, otherwise there would be unused assets. Listing 3 indicates that this was actually done; we only show part of the code associated to *LEDS_RENESAS_TPU* for a matter of space, but the *leds-renesas-tpu.c* and *leds-renesas-tpu.h* files were entirely removed from the repository.

With these deletions, products that had feature *LEDS_RENESAS_TPU* will present different behavior, unless the *PWM_RENESAS_TPU* feature has a compatible behavior to the previous one and the products having the former also had the latter, but this may not be true. Thus, in the new product line, we likely will not find products that match the behavior of a product with *LEDS_RENESAS_TPU*. Consequently, this is not considered a safe evolution scenario; the existing theory fails to support developer in this case, even though we know that products that do not have that feature should have the same behavior. In fact, this scenario is partially safe considering the configurations corresponding to the products that did not have *LEDS_RENESAS_TPU* and are not impacted by its removal. Since Linux users can choose to select or not *LEDS_RENESAS_TPU*, there might be a number of products that do not have it. Supposing that 50% of the products have *LEDS_RENESAS_TPU*, we could give support for half of the products, which would make the gain significant by avoiding, for instance, these products to be tested.

There are many other kinds of partially safe evolution scenarios, such as asset additions and in these cases, both implementation files and the respective mappings are added to the product line. In this scenario, products that suffer additions do not preserve behavior, but the evolution is still safe considering only products that do not have the added files. The percentage of refined products is directly proportional to the frequency of the respective features. If the affected feature is mandatory, the guarantee tends not to be high, since this feature possibly appears in all products (in this case we would give no guarantee). In contrast, when the changed feature is optional and positioned just near the root feature, for instance, the guarantee can achieve 50% of the products, since no more than 50% of the valid products have the respective feature, and this percentage increases when the feature is positioned lower in the three. Therefore, we believe that one could benefit from a notion of partially safe evolution, that is able to handle unsafe evolution scenarios, while still offering safe evolution guarantees considering a subset of the products.

3. PARTIALLY SAFE EVOLUTION

To handle evolution scenarios such as the one illustrated in the motivating example, we introduce a partial refinement theory that formalizes our notion of partially safe evolution of product lines. Moreover, we also present some properties and analyze how partial and full refinement operations can be interleaved, which might be often necessary in practice.

3.1 Partial Refinement

To define the partial refinement notion, we rely on existing concepts from the refinement theory [1, 3]. A product line is defined as a well-formed triple: a feature model (FM) that has features and dependencies among them; an asset mapping (AM) that relates asset names and assets; a configuration knowledge (CK) that maps features to assets. We do not require specific languages for these elements. The FM, for instance, could be a variability model (VM), such as the Linux Kconfig. For an arbitrary FM F, we assume a semantics function $[\![F]\!]$, which returns the set of all valid configurations generated from F. A configuration is a feature selection, which can usually be represented as a set of feature names. There is also a semantics function for the CK, denoted by $[\![K]\!]^A_c$, that takes a CK K, an AM A, a configuration c and yields the respective product. In practice, users select the desired features that constitute a configuration. By processing the CK, it is then possible to obtain the assets that constitute the product represented by a configuration. A product is defined as a set of assets, and a product p' refines another product p, denoted by $p \sqsubseteq p'$, whenever p' it at least as good as p, in the sense that it preserves the observable behavior of p [3].

Product line refinement happens when all products in the original product line are refined in the evolved product line, as established in Definition 1. This applies when locally refactoring code, or removing unused assets, for example. We should notice that the definition only requires product refinement to hold, therefore configurations are allowed to change when matching a product of the original product line with a product of the new product line. Thus, feature renaming is a full refinement, as feature names do not matter.

Definition 1 (Product line refinement)**.** For arbitrary product lines $L = (F, A, K)$ and $L' = (F', A', K')$, L' refines L, denoted by $L \sqsubseteq L'$, whenever

$$\forall c \in [\![F]\!] \cdot \exists c' \in [\![F']\!] \cdot [\![K]\!]^A_c \sqsubseteq [\![K']\!]^{A'}_{c'}.$$

Contrasting, the partial notion requires that only some products are refined. So, in Definition 2, we use S as an index to denote the subset of refined product configurations. More precisely, for product lines L and L', and set of configurations S, we say that L' partially refines L for the configurations in S when S is in both FMs, and product refinement holds for all configurations that are in S. The first condition is necessary to guarantee that all configurations in S are valid according to the respective product line. Otherwise, we would not be able to generate valid products.

Definition 2 (Partial product line refinement)**.** For arbitrary product lines $L = (F, A, K)$ and $L' = (F', A', K')$, and a set of configurations S, L' partially refines L for the configurations in S, denoted by $L \sqsubseteq_S L'$, whenever

$$S \subseteq [\![F]\!] \wedge S \subseteq [\![F']\!] \wedge \forall c \in S \cdot [\![K]\!]^A_c \sqsubseteq [\![K']\!]^{A'}_c.$$

With this relation, we can now support developers in examples like the one in Section 2. We could simply associate L with the product line before the feature removal, and L' represents the modified product line without the *LEDS_RENESAS_TPU* feature. Thus, S would be the set of all configurations that do not contain *LEDS_RENESAS_TPU*. This would include configurations such as $\{IMX\_WEIM, MVEBU\_MBUS, OMAP\_OCP2SCP, ...\}$ and $\{ADB\_IOP, ADB\_MACII, PROC\_EVENTS, ...\}$. Since the only modification is the feature removal, and we filter the respective changed products by verifying refinement only for configurations in S, partial refinement holds. Hence, developers would only need to test products that had *LEDS_RENESAS_TPU*, which could consequently

increase productivity. The previous theory gives no guarantees for this case, so developers would have no support nor guarantees.

We should observe that in Definition 2 the configuration c remains the same, so feature names do matter. Moreover, we only check refinement for products in the scope of S. This means that the larger the S, the better is the support. We could try to provide some guarantee for products that are not in the scope of S. However, this would involve having a notion of partial product refinement, and this is not in the scope of this work. It is also the case that, for an empty S, partial refinement trivially holds. However, this means that we would give no support, so developers will have no benefit in establishing that.

The partial refinement relation is reflexive and transitive, which are essential conditions to support stepwise partially safe evolution. Theorem 1 establishes that every product line is partially refined by itself. As required by Definition 2, we need to assure that S is a subset of the valid configurations generated from the respective product line.

Theorem 1 (Partial product line refinement reflexivity)**.** For a product line L and a set of configurations S, let F be the FM of L. If $S \subseteq [\![F]\!]$, then $L \sqsubseteq_S L$.

One might want to consecutively perform partial refinement operations, and the transitivity property guarantees that this is feasible, and that it could still result in some refined products. However, given that the partial refinements might involve different subsets of products, we can only guarantee that refinement holds for the intersection of the configurations refined in each step. For instance, given a product line $L1$, one could first fix a bug, obtaining a product line $L2$, and then remove a feature, obtaining $L3$. Assuming that S and T are the sets of configurations refined in each step, S would be the set of configurations whose products do not contain the changed files in the bug fix, and T would be the set of configurations that do not have the removed feature. The resulting product line $L3$ does not partially refines $L1$ in terms of S or T in isolation, because the products refined in the first step are not necessarily refined in the second step, and vice versa. But $L3$ partially refines $L1$ for the configurations that are in both sets: $S \cap T$.

Theorem 2 (Partial product line refinement transitivity)**.** For product lines $L1$, $L2$ and $L3$, and sets of configurations S and T, if $L1 \sqsubseteq_S L2$ and $L2 \sqsubseteq_T L3$, then $L1 \sqsubseteq_{S \cap T} L3$.

3.2 Compositionality

Product lines are composed of three elements, which may evolve separately to be later integrated to generate products. In this context, one might need to change a specific artifact, for instance, the FM, without changing the AM and CK. Developers could also modify different elements of the product line. We analyze these scenarios and whether such modifications preserve product line partial refinement. Compositionality theorems are provided in the existing full refinement theory, so it would be important to provide the same kind of modular support for partial refinement too.

It is important to notice that FM equivalence and FM refinement lead to product line partial refinement only if S is derived from the initial FM. Thus, we establish a weaker FM equivalence notion, which allows the initial FM to have configurations not derived from the final FM, differently from the FM equivalence and refinement notions, that require the semantics of the initial FM to be equal or a subset of the semantics of the final FM [3]. According to Definition 3, the FMs only have a set of configurations S in common. If we change a feature from optional to mandatory, for instance, weaker equivalence holds. In this scenario, S would be the set of configurations that already had the changed feature.

Definition 3 (Feature model weaker equivalence)**.** For arbitrary feature models F and F', and a set of configurations S, F is equivalent to F' in terms of S, denoted by $F \cong_S F'$, whenever

$$\forall c \in S \cdot c \in [\![F]\!] \wedge c \in [\![F']\!].$$

As captured in Theorem 3, FM weaker equivalence preserves product line partial refinement. Given a product line L, one can modify the FM only, by possibly adding, removing or modifying features and dependencies among them, but preserving a set of configurations S. Whenever only the FM is changed, there is still a partial product line refinement with respect to the same S. Since a product line by definition is well-formed [3], we know that L is well-formed. However, we have no guarantee about L', more precisely, whether configurations that are in F' but are not in S lead to valid products. In exceptional cases, these could refer to features that are not in F' or do not obey rules, such as having a feature without having its parent. This is the reason for requiring well-formedness. Refinement holds because we are not checking products whose configurations are not in S. Moreover, the weaker FM equivalence guarantees that S is in both FMs. Neither the AM nor the CK change. Therefore, we actually have exactly the same products if only checking configurations from S.

Theorem 3 (Feature model weaker equivalence compositionality)**.** For a product line $L = (F, A, K)$, a feature model F', and a set of configurations S, let $L' = (F', A, K)$. If $F \cong_S F'$ and L' is well-formed, then $L \sqsubseteq_S L'$.

Defining partial refinement relations for the AM and CK elements is part of our future work. We suspect that they would also lead to product line partial refinement, as the FM weaker equivalence does.

3.3 Combining full and partial refinement

We also reason about compositionality in terms of combining different refinement notions, since the theories are not mutually exclusive; they are complementary. Thus, practitioners may desire to perform partial and full refinement operations together. For improvements or adding new features with behavior preservation, one can make use of the full refinement theory. After that, developers may need to later remove another feature, such as the feature removal scenario illustrated in Section 2. For these cases, the partial refinement notion should be more appropriate. Hence, the theories might be used interchangeably and we need to provide support, in the sense that when applying consecutive transformations, refinement still holds for some products.

Full and partial refinement

When a partial refinement is followed by a full refinement, we would ideally have partial refinement for products in S by transitivity. This is not possible because, in the full refinement transformation, feature names do not matter, differently from the partial refinement notion. In fact, as Definition 1 admits configurations to change, even when S is

equal to the set of all valid configurations, full refinement is not necessarily a particular case of partial refinement.

Definition 4 describes an alternative notion of partial refinement that allows configurations to change according to a renaming function f. Then, given an initial configuration c, refinement holds for the product generated from $f(c)$. In a feature renaming situation, supposing that we change the feature name from P to P', f would be defined as $f(c) = c[P'/P]$. Basically, configurations are the same as before, except that instead of having P we now have P'.

Definition 4 (Weak partial refinement)**.** For arbitrary product lines $L = (F, A, K)$, $L' = (F', A', K')$ and a function $f : Conf \rightarrow Conf$, L' partially refines L in terms of f, denoted by $L \sqsubseteq_f L'$, whenever

$$\forall c \in dom(f) \cdot f(c) \in [\![F']\!] \wedge [\![K]\!]^A_c \sqsubseteq [\![K']\!]^{A'}_{f(c)}.$$

The partial refinement notion is a particular case of Definition 4 (when f is the identity function over S). Thus, this weak notion supports situations where configurations change, which are not covered by the default partial product line refinement notion (Definition 2). Since the weak definition is more general, we could have it only instead of having both partial refinement relations. However, Definition 2 is less complex and developers only need to deal with the alternative one for feature renaming scenarios.

We have a function f as an index because allowing configurations to be arbitrarily modified having a set of configurations S as an index would lead to relations that are not transitive. Transitivity does not hold for such a definition because we have no control of the new configurations; they could be arbitrary. Thus, when applying consecutive refinements, we would not know if the configurations refined were the same as the one refined in the first step. Hence, even assuming two refinement operations in terms of the same S, the transitivity does not hold for S. We do not desire to have a partial refinement relation that is not a pre-order, since this would seriously limit its applicability.

When one applies a partial refinement followed by a full refinement, we have a weak partial refinement. A possible scenario of such situation is found, for instance, when instead of only renaming a feature, this operation follows an asset change in a non behavior-preserving way. Since not all products are refined because of the asset change operation, the domain of the function is only the set of configurations whose products do not have the changed asset. Supposing that L is the product line before these two operations and L' is the final product line, we then guarantee that $L \sqsubseteq_f L'$. The function f in this case would also be defined as $f(c) = c[P'/P]$, since in the asset change operation configurations were not changed. This notion is formalized in Theorem 4. When a partial refinement is followed by a full refinement, there is a function that maps configurations from S to the final product line, so that the weaker partial refinement holds.

Theorem 4 (Partial and full refinement)**.** For product lines $L1$, $L2$ and $L3$ and a set of configurations S, let $F3$ be the FM of $L3$. If $L1 \sqsubseteq_S L2$ and $L2 \sqsubseteq L3$, then, for some function $f : S \rightarrow [\![F3]\!]$, $L1 \sqsubseteq_f L3$.

If the operations are conducted in the opposite order (full refinement followed by partial refinement), the reasoning and end result are analogous, so we omit the details here.

Stronger full and partial refinement

Although the full and partial refinement correspondence is not direct, there is a stronger definition for product line refinement that has basically the same meaning of Definition 1, but it does not allow changes in configurations. Consequently, this definition gives support to less scenarios when comparing to the other one. Feature renaming, for instance, is not a refinement according to this notion. Since configurations are sets of feature names, when changing such names, configurations containing them are impacted.

Definition 5 (Stronger product line refinement)**.** For arbitrary product lines $L = (F, A, K)$ and $L' = (F', A', K')$, L' strictly refines L, denoted by $L \preceq L'$, whenever

$$\forall c \in [\![F]\!] \cdot c \in [\![F']\!] \wedge [\![K]\!]^A_c \sqsubseteq [\![K']\!]^{A'}_c.$$

Differently from product line refinement, the stronger notion is more similar to the partial refinement notion (Definition 2). Since it does not allow any change in configurations, we can then establish a more direct relationship. For product lines L and L', the stronger full refinement is always a partial refinement, provided that the set of configurations S is present in L. As a consequence, by transitivity, when a partial refinement is followed by a stronger full refinement, this results in a partial refinement, as shown in Theorem 5. If the refinements are performed in the opposite order, the result is also a partial refinement.

Theorem 5 (Partial and stronger full refinement)**.** For product lines $L1$, $L2$ and $L3$ and set of configurations S, if $L1 \sqsubseteq_S L2$ and $L2 \preceq L3$, then $L1 \sqsubseteq_S L3$.

Commutativity of full and partial refinement

Finally, we also reason whether the stronger full refinement and partial refinement transformations lead to the same product line when applied in different orders, and we demonstrate that this property holds. For instance, given a product line $L1$, suppose that users perform a stronger full refinement, such as locally refactoring an asset, obtaining $L2$ and then partially refine the product line by removing a feature, obtaining $L4$. Figure 1 represents a commutative diagram that shows that if we first apply this same partial refinement operation (yielding $L3$) and then refining the asset we obtain the same $L4$. Thus, in this case, the order in which the transformations are applied does not matter.

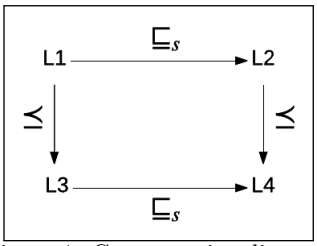

Figure 1: Commutative diagram

Properties like this one reflect what happens during development, where practitioners might want to apply several different operations consecutively and it is helpful to be sure that applying refinements in a different order produce the same result. We formally derive and prove two theorems from the commutative diagram structure shown in Figure 1.

In Theorem 6, we give support in case developers are doing first a partial refinement and then a strong full refinement.

The theorem establishes that there is an alternative way to obtain the same resulting product line, that would be doing the corresponding operations in the opposite order. Theorem 7 is analogous. This theorem has an extra condition when compared to the first one. This diagram only holds if S is a subset of the valid configurations generated by the initial product line $L1$. This condition is necessary, as otherwise we could have invalid products, since invalid configurations may not obey dependency rules among features. Thus, it does not make sense to refine a product line in terms of an S that is not part of the product line configurations.

Theorem 6 (Commutative diagram (1))**.** For product lines $L1$, $L2$ and $L4$, and a set of configurations S, if $L1 \sqsubseteq_S L2$ and $L2 \preceq L4$, then, for some product line $L3$, we have $L1 \preceq L3 \wedge L3 \sqsubseteq_S L4$.

Theorem 7 (Commutative diagram (2))**.** For product lines $L1$, $L3$, $L4$ and a set of configurations S. Let $F1$ be the FM of $L1$. If $S \subseteq [\![F1]\!]$, $L1 \preceq L3$ and $L3 \sqsubseteq_S L4$, then, for some product line $L2$, we have $L1 \sqsubseteq_S L2 \wedge L2 \preceq L4$.

For space restrictions, we omit all proofs. Also, in Section 4 and Section 5, we present a restricted number of templates and queries. The full data can be found online.[6]

4. PARTIALLY SAFE EVOLUTION TEMPLATES

As mentioned in Section 3, the partial refinement theory can be applied to several contexts. In this section, we exemplify such contexts and define templates that are abstractions of practical evolution scenarios. Templates are helpful because they free developers from understanding the theory; the templates are easier to understand and provide guidance on how to evolve a product line guaranteeing safe evolution for a subset of the products. Additionally, they avoid errors during an evolution process and increase developers confidence. A template has a left-hand side pattern (LHS) and a right-hand side pattern (RHS). They correspond to abstractions that capture properties of the initial and evolved product lines, respectively. In case one follows the syntactic and semantic rules established by templates, it is guaranteed that partial refinement holds for a specified subset of products S. The developer does not choose the value of S; it is defined based on the FM, AM and CK of the product lines in the templates. Establishing S this way helps to understand the impact of the change, since products in the scope of S are not impacted.

Remove feature

We first analyze feature removal situations, which is an usual scenario in a product line development context. One often decides to exclude features for diverse reasons [15]; for instance, they are no longer used or not needed by customers. We then define a REMOVE FEATURE template in Figure 2. For these cases, products that did not have the removed feature in the original product line keep the same behavior, and the others might not be refined. In this template, the three product line elements are changed. By syntactically analyzing the REMOVE FEATURE template in Figure 2, we observe that the initial FM, F, has a feature O to be removed, and consequently, F' does not have it. We also notice that O is descendant of P. Nothing else is changed in the FM, which might have other features beyond the required O and P. We assume that the initial CK has references to O, so from the LHS to the RHS, the CK line (containing e' and n') referencing O is removed. The AM also loses a mapping, which represents the asset that implements O.

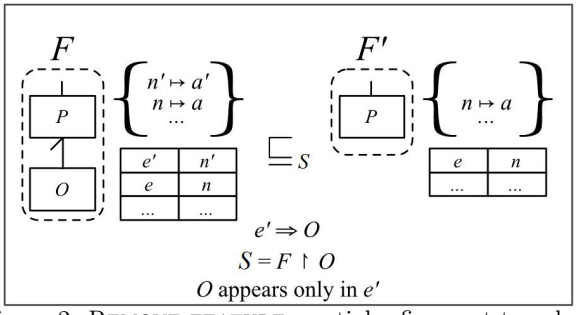

Figure 2: REMOVE FEATURE partial refinement template

The guarantees provided by the template only hold if some conditions are valid. We need to make sure that when e' is true, O has been selected, otherwise it would make no sense to exclude it from the CK. To do so, we require that $e' \Rightarrow O$. Consequently, a' is removed, since it must be in a product whenever O is selected. When a feature is removed, the intuition is that the products that did not have the respective feature do not change behavior. For the other ones, refinement does not hold because they lose functionality unless a' adds no extra behavior to a product. To specify S to capture that, we make use of the $\upharpoonright$ operator, which filters configurations from a FM according to a feature expression. This expression may contain feature names and logical operators, such as *and*, *not*. The expression $P \wedge Q$, for instance, is satisfied by a configuration c when c has features P and Q. For an arbitrary FM F and a feature expression e, we use $F \upharpoonright e$ to denote the set of configurations in $[\![F]\!]$ and that do not satisfy e. Thus, we specify S as $F \upharpoonright O$, giving refinement guarantees only for product configurations that are in F and do not include O. Since we only remove the line containing e' and n' from the CK, it is required that O does not appear in e and other CK lines, otherwise the feature would not be completely removed.

This template matches the example discussed in Section 2. To illustrate that, we instantiate the meta-variables for the example. In this case, F is instantiated with the initial Linux VM containing *LEDS_RENESAS_TPU*, and F' is the resultant VM without this feature. The initial CK is instantiated with the Linux CK, including the line shown in Listing 2 and the changed CK is the same, but without this mapping. The Linux AM could be represented by mappings between the asset names to their respective contents. Using the feature removal example, n' would be *drivers/leds/leds-renesas-tpu.c* and *drivers/leds/leds-renesas-tpu.h*, and a', the respective contents of these source code files. The other mappings, such as $n \mapsto a$, correspond to other source file names and the respective contents. The new AM is obtained from the initial by removing the mapping $n' \mapsto a'$, which corresponds to the implementation of the removed feature. It is true that $e' \Rightarrow O$, since e' is *LEDS_RENESAS_TPU*. This feature appears only in e', since we did not find occurrences of this feature in the remaining items of the CK. S is $F \upharpoonright$ *LEDS_RENESAS_TPU*. Thus, for these configurations the refinement holds. The other products are not refined since

[6] http://github.com/spgroup/theory-pl-refinement

they have the removed feature, thus not preserving behavior.

Change asset

Another situation we analyze is an asset change. Developers modify source files in many contexts, such as fixing bugs or implementing new features. In such situations, one possibly not desires to preserve behavior. Thus, this is often not a safe evolution scenario, since products that contain the changed asset might not preserve behavior. Therefore, we give refinement guarantees for the other products, which are the ones that do not have the changed assets. We define a template that matches this scenario in Figure 3.

To specify S for this case, we use another restriction operator. For arbitrary FM F, CK K and set of asset names ns, we use $(F, K) \upharpoonright ns$ to denote the set of configurations in F semantics but whose features are not present in products containing assets from ns. Hence, S is defined as $(F, K) \upharpoonright \{n\}$, which is the set of configurations that are in F whose features are not implemented by the asset named n, which in this case is the a asset. Since products containing a' are not refined, we can not give any guarantees for them. There is also a well-formedness condition. Since we do not know which changes were performed to a, and we define product lines as well-formed triples, we need to demand well-formedness for products containing a'.

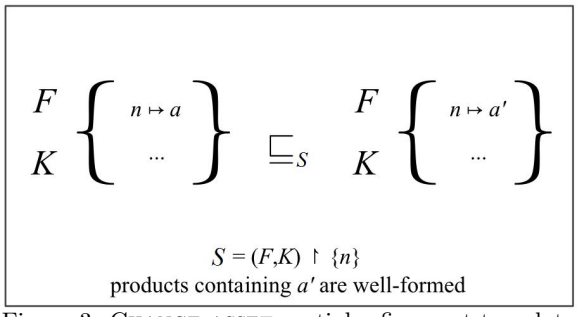

Figure 3: Change asset partial refinement template

It is possible to notice that in some of these scenarios, not all product line elements change. The Change asset template assumes that both product lines have the same F and K. More precisely, only the asset a is changed to a'. Thus, only the asset content is modified, not the asset name, which is the same for the initial and new lines (n). Although this template does not capture situations where the FM and CK change as well, one could obtain this effect by combining templates. The Change asset template can be used with the Change ck line template (which will be introduced later), for instance. Thus, developers could not only change assets, but also change their reference in the CK. As explained in Section 3, the guarantee is for the intersection of the products refined in both steps and this can be automatically calculated, as we define the S for both templates.

We capture not only unsafe evolution scenarios with the Change asset template. Supposing that one might refine an asset, this template also matches. However, we would give less support since we assume that the asset is being changed in a non behavior-preserving way. Thus, the Refine asset template [12] is more appropriate in this situation because it assumes that the product line is safely evolved and gives guarantees for all products. In contrast, if the change impacts the product line behavior, the Refine asset template gives no support and developers should rather make use of the Change asset template.

Add assets

Another possibility of partially evolving a product line is by adding new assets and their references in the CK. This is possibly not a safe evolution scenario, since some features might be associated to new assets that change their behavior. Thus, we characterize it as a partially safe evolution scenario and give refinement guarantees only for products that are not extended with the new assets. The template illustrated in Figure 4 deals with this situation. It requires the FM not to change. We now use meta-variables to represent the entire product lines. This is just to reference them in the template conditions. The AM in PL' is an extension of the previous AM with new mappings, and new items are added to the CK. However, nothing is removed. We are only assuming additions to both entities.

The set of refined configurations in this case is similar to the previous template, but instead of filtering according to a single asset name n, we are filtering products that do not have names present in $dom(m)$, where $dom(m)$ is the left side of m; these are precisely the products that were not extended. Another condition is that the assets in its must appear in m. This is necessary, otherwise we could still have non-refined products after filtering the configurations that compose S. We also require that products from PL' that are not in S to be well-formed. Since the AM and CK are modified, we have no control of such additions, so we demand well-formedness.

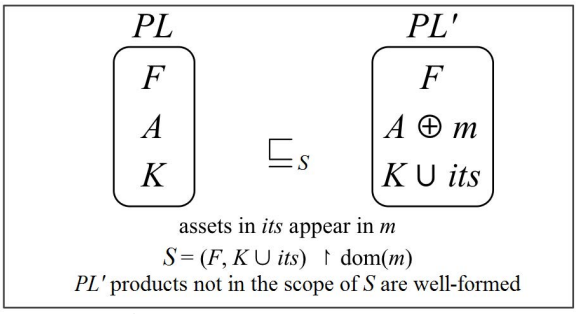

Figure 4: Add assets partial refinement template

This template does not require that A and K are modified, since m and its could be empty. However, in case unused assets are added (by changing only the AM), one should rather use the Add unused assets safe evolution template [12] to obtain refinement guarantees for all products. Moreover, if the three elements of the product line are modified and the assets added are only in respect to the new feature, this would also be a safe feature addition scenario [12], provided that the added feature is optional. However, in cases of non-safe feature additions, such as mandatory feature additions, one could obtain support of the partial refinement theory by dividing this modification in two parts: first, only the FM would be modified by adding a mandatory feature using the Add new mandatory feature template [12], and then adding the respective assets by using the template illustrated in Figure 4. The first step is safe, since it is a FM refinement scenario. The second step, however, is not totally safe because the mandatory feature added in the previous step had no implementation, and now part of the products are going to be modified with these new assets. The guarantee would be for products that do not have the

added feature, as guaranteed by Theorem 5.

Templates derivation process

We derived the templates by adapting a catalog of safe evolution templates [12, 3] for situations where not all products are refined by products in the evolved product line. For instance, the CHANGE ASSET template in Figure 3 essentially adapts the REFINE ASSET [12] template by dropping the precondition that the new asset a' must refine a. This way we allow any kind of change to a, but capture change impact by precisely defining S. Verify completeness of the templates and derive new ones are part of our future work.

5. EVALUATION

Although we suspected that the partially safe evolution templates could be useful in a number of situations, it is important to better understand how often they could apply in practice. To do that, we analyze 79218 evolution scenarios of the Linux repository. We basically try to find only scenarios that match our templates. Strictly, every evolution scenario that matches our templates is partially safe, if we make S empty, but it is not our goal to give no guarantees. In this section, we detail the data extraction process in Section 5.1, show the results obtained for each template in Section 5.2, and discuss threats to validity in Section 5.3. The purpose of this study is to discover whether the proposed templates could be frequently applicable in a product line development context. We would like to answer the following question:

RQ: How often are partially safe evolution templates applicable in a product line project?

In order to answer this question, we automatically analyze commits of the Linux project, where each commit corresponds to an evolution scenario composed of the product lines just before and after the commit is performed. We measure the number of occurrences of the proposed templates, since they represent partially safe evolution situations.

5.1 Data extraction

We assess the templates using FEVER.[7] This tool is able to analyze commits from the Linux repository. FEVER receives Linux versions as input and collects all commits from the respective releases. Then, the differences are processed and the resulting information is stored in a Neo4j database.[8] To find occurrences of our templates, we query the database populated by FEVER filtering evolution scenarios by expressing the conditions for each template, such as whether they touch the FM. We also manually check some evolution scenarios to make sure they really match the templates. Altogether, we analyzed 79218 evolution scenarios from the database we had access to, and this corresponds to all commits between the Linux versions 3.11 and 3.16. The first commit was performed on September 2^{nd} of 2013 and the last one was on August 3^{rd} of 2014, so this comprises roughly one year of development. We try to match each evolution scenario with the templates, based on their conditions.

REMOVE FEATURE: scenarios that modify all three elements of the product line, removing elements. These three modifications must be correlated, as exemplified in Section 2. Thus, the removed mappings need to be from the removed features in the FM. Similarly, the removed assets in the CK need to also be excluded from the implementation. These rules are detailed in Listing 4. From the *MATCH* clause, we have all commits in which the CK and source code are both changed. We then have the *WHERE* clause to filter them with extra conditions. For instance, the first condition is that this commit should affect the FM as well, and the change must be a removal. Moreover, the name of the feature in the FM needs to be the same name of the feature edited in the mapping change (CK). All distinct commits obeying these rules are then returned.

Listing 4: Remove feature Neo4j[8] query

```
1 MATCH  (file:ArtefactEdit)<--(c:commit)
      -->(mapping:MappingEdit)
2 WHERE
3   (c)-->(:FeatureEdit{change:''Remove'',
        name:mapping.feature}) AND
4   file.change=''REMOVED'' AND
5   mapping.target_change=''REMOVED'' AND
6       ...
7 return distinct c
```

CHANGE ASSET: we classified an evolution scenario as a change asset instance when neither the Linux FM nor the Linux CK changed and when at least one source file changed. It was also necessary that the commit had no source files added or removed. Therefore, we only capture cases where the only change is in source code and non-code files, such as documentation. If only a *.txt* file is modified, we do not consider it a change asset instance.

CHANGE CK LINE, ADD CK LINES and REMOVE CK LINES: we identify these templates with only one query because they are very similar and we noticed that in some cases an evolution scenario was an instance of the CHANGE CK LINE template, but the Git diff algorithm was showing it as a removal followed by an addition. Since the tool relies on this classification, we could have non-precise results, so we preferred to detect mapping changes and check manually which templates match the respective evolution scenario. For all of them, we required the implementation and FM elements not to change. We also identified the other templates in a similar way, and the results are presented next.

5.2 Results

We illustrate the results of running FEVER against the 79218 commits in Table 1. According to Dintzner et al. [8], around 80% of feature oriented changes in Linux only touch the implementation, and do not affect the FM or CK. Confirming that, the CHANGE ASSET template had the highest occurrence rate. This might be due to Linux maturity level, and also to the fine granularity of the commits observed in the analyzed period. However, asset refinements also match this pattern and, since the number of occurrences is extremely high, we could not manually verify all cases. Thus, a number of these occurrences might be full refinements. By further manual analysis in 50 instances (arbitrarily chosen between versions 3.15 and 3.16), only 7 turned out to be asset refinements. The other 43 are non-refinements and the majority of them were bug fixes. Developers fixed such bugs mostly by modifying *if-then-else* conditions. Based on this analysis, we suspect that partial refinements occur more frequently, and this makes the CHANGE ASSET template far more applicable than the REFINE ASSET template.

Each scenario is classified as compatible with no more

[7] http://github.com/NZR/SPLR-FEVER-Tool.
[8] Neo4j website http://neo4j.com.

than one template, except for the ADD CK LINES, CHANGE CK LINE and REMOVE CK LINES templates. Since they were all found with the same query, we noticed that some scenarios actually had instances of more than one from the three patterns. Thus, a scenario might be classified as an instance of both REMOVE CK LINES and ADD CK LINES templates, so we had to proceed with a manual analysis as follows. FEVER returned 202 instances of the ADD ASSETS template, of which 49 were manually checked to confirm they really match the template. Two of them did not, because they were also modifying the FM, so they were actually feature additions. These two instances were removed from Table 1. Although we specified in the query that the resultant scenarios must not touch the Linux FM, these two cases were accidentally returned by the tool. The other 47 match exactly our conditions and are instances of the template. There are at least 21 assets removals. We did not investigate the reason of for lower removal rate. The results might be different considering another interval and project.

There are at least 26, 15 and 11 scenarios respectively corresponding to mapping changes, additions and removals. We manually checked the 52 instances. These numbers are not very high because modifications focusing only on the mapping rarely occur, as the templates CHANGE CK LINE, ADD CK LINES and REMOVE CK LINES have a low frequency when comparing to others, such as ADD ASSETS. This might happen because most of the commits modify at least one source code file and some of them modify also the FM. The CHANGE CK LINE template presents the highest number of instances of the three patterns, probably because developers often remove and add mappings together with the respective source code associated or references to the FM as well. It is also possible that an evolution scenario captured by one of our templates corresponds to many commits. Since we try to match each commit separately with the templates, this would explain the low occurrence.

We confirmed that 59 of the 92 feature removal scenarios match our REMOVE FEATURE template. Our query returned also some non-removals, such as scenarios where the features were actually being moved. We did not consider these cases to match our pattern, since the feature was not actually removed. For the 33 remaining scenarios, we read the commit messages and confirmed whether they were mentioning feature removals. For the cases that did not mention any removal, we manually analyzed and only included in our results cases that were indeed instances of the template.

Template	Instances
CHANGE ASSET (and possibly REFINE ASSET)	65129
ADD ASSETS	200
REMOVE ASSETS	21
CHANGE CK LINE	26
ADD CK LINES	15
REMOVE CK LINES	11
REMOVE FEATURE	92

Table 1: Template occurrence

As just explained, the numbers in Table 1 are lower bounds of the cases we could confirm. From the 79218 commits, 6391 are merge commits, which are discarded by the tool because they correspond to integration, not evolution, scenarios. Although there might be changes during manual merges, they are not really relative to a single previous product line, as captured by our templates. Hence, we could give support for approximately 90% of the cases. There are, in fact, 7333 commits that, together with its previous commits, do not match any of our templates, which could include, for instance, commits that only change feature dependencies in the FM, or commits that represent feature additions.

5.3 Threats to validity

As this is a preliminary evaluation, in this section, we discuss internal, external and construct validity.

Construct: As already mentioned in Section 5.2, to find occurrences of the CHANGE ASSET template, we search for any change in the implementation and do not analyze which type of modification was performed, thus also having commits which actually represent occurrences of the REFINE ASSET template [12]. We manually examined 50 commits, so we can not generalize for the 65219 occurrences. Scenarios matching the other templates can be safe only in abnormal cases, so we do not take them into consideration.

Internal: We should consider that the tool we use may have bugs and false positives/negatives. We manually analyzed several instances, so there is evidence that, except for change asset instances, most of the results returned are correct. We did not find false negatives, but if we could detect and reduce them, we would have even better results. False negatives rate also depends on the number of commits matched to an evolution scenario. We analyzed each commit separately. For instance, one could remove a feature in two parts: first, the FM and CK could be changed, and in the subsequent commit only the implementation would be removed. In this situation, these two commits would not match any template, although, in sequence, they constitute a feature removal scenario. We also consider as internal validity the queries precision, since it is not trivial to precisely classify commit changes.

External: We only examined a small part of the Linux repository history. Hence, we can not generalize the result for other history periods or projects, which may have different development practices, such as commits with coarser granularity and different programming languages. Perhaps, if we analyze other projects, the CHANGE ASSET template could be used together with others, since one might change not only the implementation but also the FM and the CK in a single commit. However, as a consequence, other templates could have a higher rate of occurrence. Although we do not include other projects, we consider the Linux system significant because of its popularity and complexity.

6. RELATED WORK

Dintzner et al. [7] present a classification of feature changes as well as a tool named *FMDiff* to automatically analyze differences in Linux variability models. The change categories are specific to structures found in Kconfig specifications, such as feature dependency changes. Finally, they evaluate the tool by analyzing commits from the Linux repository history. Thüm et al. [18] classify FM edits into refactorings, specializations, generalizations and arbitrary edits by using satisfiability solvers. Our work differs because it is not our goal at the moment to build a tool and to analyze the feature model structure only. However, our theory could be mechanized in such tools to provide even more support for developers when making changes to the FM, by providing the subset of refined configurations in each case.

Passos et al. [15] propose a pattern catalog containing feature addition and removal templates applicable in the Linux context. The main difference from their patterns to ours is that they do not focus on giving guarantees for developers in partially safe evolution scenarios. Additionally, they present both refinement and potential non-refinement templates. To verify the scenarios occurrence in practice, they conducted an experiment by manually analyzing the Linux repository trying to find instances of their templates and discarded the ones that did not present a significant occurrence rate. While they focus on proposing templates not only representing refinement scenarios but also non-refinements, our aim is to propose a new refinement theory and non-full-refinement templates. They also suggest the need for a new theory to address non-full-refinement scenarios.

Nieke et al. [13] analyze feature model evolution and define temporal feature models, which allow features to have expiration date. For instance, if a feature is removed it is no longer valid. It is also possible to have *locked* configurations. A configuration that is *locked* should never be broken. This information is achieved through analyzing possible changes, such as feature renaming, deletion, among others, to temporal FMs. This work resembles ours because it gives support for some partial refinements regarding the variability model. Developers can change some configurations and still be certified that the *locked* ones remain valid. However, they only analyze the variability model and do not propose a partial refinement theory, differently from our work.

A number of researchers [5, 10, 11, 6] use model checking techniques [4] to verify products lines. Sabouri and Khosravi [17] try to tackle the well-known state space explosion problem by statically analyzing product family models, before checking them against properties (expressed in linear temporal logic [9]), to avoid re-verifying products by using previous results. This work is related to ours, since in both proposals, not all products are verified. However, we defined a partial refinement theory and verify product refinement, whereas their focus is not refinement, but to verify general properties, such as whether a product has a specific feature.

7. CONCLUSION

In this work, we define a partial refinement theory for product lines with the aim of covering unsafe product line evolution scenarios not supported by any previous theory. We establish connections between such theories by showing how they can be used together and suggest a template catalog which represents common partial evolution scenarios found in practice. For each template, we analyze the change impact and give guarantee of refinement for a proper subset of the original products. Finally, we gather template occurrence evidence in the Linux project.

As future work, we plan to formalize partial refinement notions for the AM and CK artifacts and define compositionality theorems regarding these notions. We shall also prove more commutative properties, and this includes enlarging our refinement notion to have transformation functions relating product lines before and after evolution. We also intend to expand our empirical analysis, which is still preliminary, by evaluating projects other than Linux, and determining exactly the value of S for each scenario. We could also investigate the completeness of the templates. Finally, we would like to develop a tool to support developers on software product line evolution.

8. ACKNOWLEDGMENTS

We thank Nicolas Dintzner for the FEVER tool, and members of the SPG for comments. We also thank the support from INES, CAPES, CNPq and FACEPE.

9. REFERENCES

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

[2] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-oriented Software Product Lines: Concepts and implementation*. Springer, 2013.

[3] P. Borba, L. Teixeira, and R. Gheyi. A Theory of Software Product Line Refinement. *Theoretical Computer Science*, 455:2–30, 2012.

[4] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.

[5] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. *ICSE*, 2010.

[6] M. Cordy, A. Classen, G. Perrouin, P.-Y. Schobbens, P. Heymans, and A. Legay. Simulation-based abstractions for software product-line model checking. *ICSE*, 2012.

[7] N. Dintzner, A. Van Deursen, and M. Pinzger. Extracting feature model changes from the Linux kernel using FMDiff. *VaMoS*, 2016.

[8] N. Dintzner, A. van Deursen, and M. Pinzger. FEVER: Extracting Feature-oriented Changes from Commits. *MSR*, Accepted for publication, 2016.

[9] E. A. Emerson. Temporal and modal logic. *Handbook of Theoretical Computer Science, Formal Models and Sematics*, B(1072):5, 1990.

[10] A. Gruler, M. Leucker, and K. Scheidemann. Modeling and model checking software product lines. *FMOODS*, 2008.

[11] K. Lauenroth, K. Pohl, and S. Toehning. Model checking of domain artifacts in product line engineering. *ASE*, 2009.

[12] L. Neves, P. Borba, V. Alves, L. Turnes, L. Teixeira, D. Sena, and U. Kulesza. Safe evolution templates for software product lines. *JSS*, 106:42–58, 2015.

[13] M. Nieke, C. Seidl, and S. Schuster. Guaranteeing Configuration Validity in Evolving Software Product Lines. *VaMoS*, 2016.

[14] S. Owre, N. Shankar, J. M. Rushby, and D. W. Stringer-Calvert. PVS Language Reference. *SRI International*, 2001. Version 2.4.

[15] L. Passos, L. Teixeira, N. Dintzner, S. Apel, A. Wąsowski, K. Czarnecki, P. Borba, and J. Guo. Coevolution of variability models and related software artifacts. *Empirical Software Engineering*, 2015.

[16] K. Pohl, G. Böckle, and F. J. van Der Linden. *Software product line engineering: foundations, principles and techniques*. Springer, 2005.

[17] H. Sabouri and R. Khosravi. Reducing the verification cost of evolving product families using static analysis techniques. *Science of Computer Programming*, 2014.

[18] T. Thüm, D. Batory, and C. Kästner. Reasoning about edits to feature models. *ICSE*, 2009.