The Journal of Systems and Software 155 (2019) 17–42

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Partially safe evolution of software product lines

Gabriela Sampaio*, Paulo Borba, Leopoldo Teixeira

Informatics Center, Federal University of Pernambuco, Av. Jornalista Anibal Fernandes, Recife 50740-560, PE, Brazil

A R T I C L E   I N F O

Article history:
Received 10 March 2018
Revised 12 February 2019
Accepted 16 April 2019
Available online 23 April 2019

Keywords:
Product line evolution
Product line maintenance
Product line refinement

A B S T R A C T

Software Product Lines allow the automatic generation of related products built with reusable artefacts. In this context, developers may need to perform changes and check whether products are affected. A strategy to perform such analysis is verifying behaviour preservation through the use of formal theories. The product line refinement notion requires behaviour preservation for all existing products. Nevertheless, in evolution scenarios like bug fixes, some products intentionally have their behaviour changed. To support developers in these and other unsafe scenarios, we define a theory of partial product line refinement that helps to precisely understand which products are affected by a change. This provides a kind of impact analysis that could, for example, reduce test effort, since only affected products need to be tested. We provide properties such as compositionality, which deals with changes to a specific product line element, and general properties to support developers when safe and partially safe scenarios are combined. We also define a set of transformation templates, which are classified according to their compatibility to specific types of product lines. To evaluate our work, we analyse two product lines: Linux and Soletta, to discover if our templates could be helpful in evolving these systems.

1. Introduction

Product lines provide systematic reuse and mass customisation for software related products (Clements and Northrop, 2001; Pohl et al., 2005). This concept brings advantages such as productivity and quality improvements, apart from the capacity to customise a system based on customers needs. Nevertheless, there are several challenges in the product line development field. Due to requirement changes, they naturally evolve and tend to become complex to manage. So developers face the challenge, for example, of guaranteeing that evolution is safe and users are not inadvertently affected in an evolution scenario (Apel et al., 2013; Pohl et al., 2005).

This safe evolution concept (Neves et al., 2015) is formalised by a refinement notion (Borba et al., 2012) that requires every product of the initial product line to have compatible behaviour with at least one product of the newly evolved product line. This is useful to support developers in making 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 (Borba et al., 2012; Neves et al., 2015). 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 behaviour preservation for all products. Nevertheless, we believe that we could still support developers even when all-products constraint does not apply. For example, adding functionality to an asset changes the behaviour of all products that use that asset. However, products that do not use the modified asset should not be affected. So we could still provide behaviour preserving guarantees for a proper subset of the products in the 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 behaviour preservation of the refined products, focusing only on testing the new functionality in the subset of products 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 concept can be helpful not only in a practical product line development context, but also in building tools that support product line development.

In fact, many evolution scenarios found in practice do not characterise refinements. Changing a top level (child of root) feature from optional to mandatory, for example, is not refinement

* Corresponding author.
E-mail addresses: gcs@cin.ufpe.br (G. Sampaio), phmb@cin.ufpe.br (P. Borba), lmt@cin.ufpe.br (L. Teixeira).

https://doi.org/10.1016/j.jss.2019.04.051
0164-1212/© 2019 Elsevier Inc. All rights reserved.

because not all original products are refined. More specifically, products that already had the changed feature preserve the same behaviour in the new product line. However, products that did not have the feature do not preserve behaviour because, in the new product line, they will present the extra behaviour associated to the changed feature. Furthermore, Passos et al. (2015) examined commits of the Linux repository history,[1] and found that feature removals, which are not refinements unless the feature is dead or has void behaviour, often occur. The partially safe evolution notion 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.

Analogously to the fact that we formalise safe evolution in terms of a refinement notion (Borba et al., 2012; Neves et al., 2015), partially safe product line evolution is formalised in terms of a partial refinement notion. As discussed, partially safe evolution only requires behaviour preservation for a subset of the existing products in a product line.[2] We also provide a set of properties to support developers in partial refinement scenarios. For instance, to justify stepwise evolution support reasoning about the set of refined products after changes to a single product line element.

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 original refinement notion (Borba et al., 2012). Hence, they can choose a specific notion depending on the situation. Evolution in practice often interleaves different kinds of changes, ranging from refinement to partial refinement scenarios. For this reason, to support practitioners, we derive 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 transformational templates representing abstractions of recurring 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 change impact analysis.

To evaluate the applicability of our templates, we use the FEVER tool (Dintzner et al., 2017) to automatically analyse evolution scenarios found among versions 3.11 and 3.16 of the Linux Kernel repository. We also analyse commits from Soletta,[3] which is a framework for making IoT devices. We find, in both projects, a number of instances of most templates in the commit history of both projects and confirm that they could have been applied, thus reinforcing the applicability of our templates. We also formalise the concepts and prove properties and soundness of the templates in the Prototype Verification System (PVS) theorem prover (Owre et al., 2001. Version 2.4).

In summary, with the aim of giving better support for developers in partially safe evolution scenarios, in this paper we define new properties, propose and formalise templates, and enhance our evaluation by considering more evolution scenarios, including those from a new product line project. We also provide proofs for several theorems. So this article extends our previous conference paper (Sampaio et al., 2016) in four main ways:

- further properties: we provide compositionality properties for asset mapping (Section 3.2.2) and configuration knowledge (Section 3.2.3).
- theorem proofs: we include proofs encoded in the PVS theorem prover. We use a more readable language than the PVS notation, but the.pvs files can be found in our appendix (Partial refinement theory website). We also discuss the structure of our encoding (Section 5).
- template compatibility analysis: we analyse the compatibility of our templates with existing configuration knowledge (CK) languages (compositional and transformational). A CK is compositional when features are mapped to artefacts. In the transformational one, features are mapped to transformations, such as *preprocess*, which provides more flexibility to the developer. Moreover, we present templates to deal with transformational CKs (Section 4.2), going beyond the compositional ones that had been published before.
- deeper evaluation: we now analyse the Soletta project (Section 6.2.2), another product line which aims to support the development of IoT applications. Moreover, we extend our Linux evaluation by providing further information regarding the analysed scenarios (Section 6.2.1). For instance, for the Change Asset scenarios we make use of auxiliary tools to have a better understanding of the types of changes performed by developers.

This paper is organised as follows. In Section 2, we present a motivating example from the Linux repository. We introduce the required background and the partial refinement theory in Section 3, relating it with the refinement theory. In Section 4, we present a template catalogue. We present evaluation results and related work in Section 6 and Section 7, respectively. Finally, we conclude in Section 8.

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. The lines in red were removed in the commit.

In Listing 1, we observe changes to a Linux Kconfig file,[5] which plays a similar role to feature models and other variability models, establishing the product line configuration space. 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 4 and 5. 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.

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

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

[2] We use the “partial refinement” term to denote the new refinement notion, which requires refinement only for a subset of the original products in a product line.

[3] Soletta is available at http://github.com/solettaproject/soletta.

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

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

```
  config LEDS_ASIC3
         ...
- config LEDS_RENESAS_TPU
-   bool ''LED support for Renesas TPU''
-   depends on LEDS_CLASS=y && HAVE_CLK && GPIOLIB
-   help
-   ...
  config LEDS_TCA6507
         ...
```

Listing 1. drivers/leds/Kconfig.

```
 obj-$(CONFIG_LEDS_ASIC3)          += leds-asic3.o
-obj-$(CONFIG_LEDS_RENESAS_TPU)     += leds-renesas-tpu.o
 obj-$(CONFIG_LEDS_MAX8997)        += leds-max8997.o
```

Listing 2. drivers/leds/Makefile.

```
-#include ...
- ...
-MODULE_LICENSE(''GPL v2'');
```

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

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 of the file *leds-renesas-tpu.c* for brevity, but it was removed from the repository.

With these deletions, products that had the *LEDS_RENESAS_TPU* feature present different behaviour, unless the *PWM_RENESAS_TPU* feature has a compatible behaviour 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 behaviour of a product with *LEDS_RENESAS_TPU*. Consequently, this is not a safe evolution scenario; the existing product line refinement theory fails to support developers in this case, even though we know that products not having that feature should have the same behaviour. In fact, this scenario is partially safe considering the configurations corresponding to products that did not have *LEDS_RENESAS_TPU* and are, therefore, 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. If this feature were directly connected to the root, we could give support for half of the products, which would make the gain significant by avoiding, for instance, to test these products.

There are many other kinds of partially safe evolution scenarios, such as adding functionality to existing features. 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 behaviour, but the evolution is partially safe as products that do not have the added functionality are not affected by the change, and thus, preserve their behaviour. The percentage of refined products is directly proportional to the frequency of the features that have not changed. If the affected feature is mandatory, the guarantee might be weak or even void, in case of a top level feature, which is included in all products. 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; this percentage increases when the feature is positioned lower in the three. Therefore, we believe that product line engineers could benefit from a notion of partially safe evolution 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 formalises our notion of partially safe evolution of product lines. Moreover, we present properties and analyse how refinement and partial refinement operations can be interleaved, which might be often necessary in practice.

To define the partial refinement notion, we rely on existing concepts from the refinement theory (Alves et al., 2006; Borba et al., 2012). We assume a well-fordmeness function for asset sets (formalised by $wf(as)$, where as is a set of assets). We use $wf(a)$ for a single asset a to denote $wf(\{a\})$. Well-formedness could mean, for instance, that the artefacts are compiling properly. We assume this function, instead of concretely defining it, because its implementation could change depending on the particular language used for the assets of a product line. A product is then defined as a well-formed set of implementation assets, and a set of assets as' refines another set of assets as, denoted by $as \sqsubseteq as'$, whenever as' preserves the observable behaviour of as Borba et al. (2012). Assets might be available to several products (domain engineering) or local to a single product (application engineering). Our theory is focused on domain engineering.

We also assume that the asset refinement relation $\sqsubseteq$ is a preorder. Reflexivity is essential here, and this is aligned with the idea that refinement means "equal or better". Consequently, every set of assets needs to refine itself. The fact that two sets of assets are equal imply that they have the same observable behaviour. Thus, it is considered a refinement. Transitivity also holds (Borba et al., 2012). If a set of assets as is refined by a set of assets bs, and bs is refined by the set cs, cs also refines as.

Three main elements are used in product lines: a feature model (FM) that has features and dependencies among them; an asset

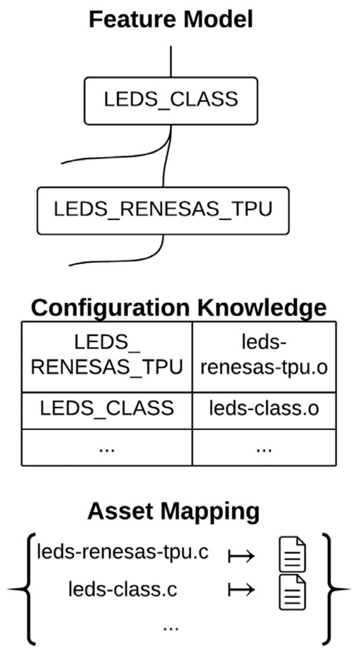

Fig. 1. SPL three main elements.

mapping (AM) that relates asset names (the AM *dom*) and assets (the AM *img*); a configuration knowledge (CK) that maps features to asset names. We provide the formal definition of the AM below (Definition 1). The CK can be defined in several ways, but just to illustrate we provide a possible concrete definition below in Definition 2. To make these definitions more clear, we illustrate these three elements in Fig. 1 by using the same example from Section 2. A product line is defined as a triple (FM,AM,CK) that generates well-formed products (Borba et al., 2012). This means that the notion of product line well-formedness is defined in terms of well-formedness for products. More specifically, a product line is considered well-formed (*wf*(*pl*)) when **all** products $p \in pl$ are well-formed (*wf*(*p*)).

We do not assume specific languages for these three elements. The FM, for instance, could be any kind of variability model, such as the Linux Kconfig. For an arbitrary FM F, we assume a semantics function $[[F]]$, which yields 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. The product generation process consists of three main phases.**(1)** Users select the desired features that constitute a configuration. **(2)** By processing the CK, it is then possible to obtain the asset names that constitute the product represented by a configuration. **(3)** The final product is obtained by checking which assets are mapped to the asset names in the AM. The three product line elements have inter-dependencies, and this makes the product line management complex. For this reason, we need to take into account these three elements in order to generate a product. The product generation function is the CK semantics function, denoted by $[[K]]_c^A$, that takes a CK K, an AM A, a configuration c and yields the respective product. When the configuration c is valid, $[[K]]_c^A$ generates a valid product.

Definition 1 (Asset Mapping)**.** An Asset Mapping *AM* is defined as a set of pairs (n, a) where n is an asset name and a is an asset. This set must satisfy the uniqueness property, which means that an asset name n is only associated with a single asset. Therefore, if there are two assets a_1 and a_2 associated with the same name (n), a_1 is equal to a_2.

$$unique(pairs) = \forall(n, a_1, a_2) : (n, a_1) \in pairs \wedge (n, a_2) \in pairs \Rightarrow a_1 = a_2$$

$$AM : \{pairs : \mathcal{F}[AssetName, Asset] \mid unique(pairs)\}$$

Definition 2 (Compositional CK)**.** A compositional *CK* is defined as a set of items, where each item is formed of a feature expression and a set of asset names. Intuitively this is a way of connecting features with their respective implementations.

$$CK : \mathcal{F}[Formula, \mathcal{P}[AssetName]]$$

Product line refinement happens when all products in the original product line are refined in the evolved product line, as established in Definition 3. 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. This happens, for example, in feature renaming scenarios, since configurations are sets of feature names, which change due to renaming. Consequently, feature renaming is a product line refinement, as feature names do not matter.

Definition 3 (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

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

3.1. Partial refinement

The refinement notion is applicable in several scenarios, such as asset refinements and feature renaming. However, there are many scenarios where a subset of the existing products do not preserve behaviour. To support examples like the one shown in Section 2, we propose a partial refinement theory.

The difference between partial refinement and refinement is that the partial notion assumes that only some products are refined and we illustrate this in Fig. 2. On the left side, the products $p1$, $p2$ and $p3$ from the initial product line $L1$ are refined by $p1'$, $p2'$ and $p3'$, respectively. The product $p4$ (in red colour) is not refined, as there is no compatible product in $L2$. This could happen in a feature removal scenario, for instance. The product $p4$ could have the removed feature, so there would be no compatible product in $L2$. We use S as the set of configurations refined, which corresponds to $\{c1, c2, c3\}$. As $p4$ is not refined, $c4$ cannot be in S. On the right side, we have product line refinement. All products from $L1$ are refined in $L2$. So, we have that $L1 \sqsubseteq L2$. Note that partial refinement holds if the configurations remain unchanged. This is the reason for having $c1$, $c2$ and $c3$ on the left side. The refinement relation, in contrast, allows configurations to change and we can have $c1'$, $c2'$, $c3'$ and $c4'$ on the new product line.

We formalise the partial refinement notion in Definition 4. We use S as an index to denote the subset of refined product configurations, that is, valid feature selections from the FM. More precisely, for product lines L and L', and set of configurations S, we say that L' partially refines L with respect to S when product configurations from S are valid for both FMs, and product refinement holds for all such configurations. The first condition is necessary to guarantee that all configurations in S are valid according to the respective product lines. Otherwise, the set S could have spurious configurations.

Definition 4 (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

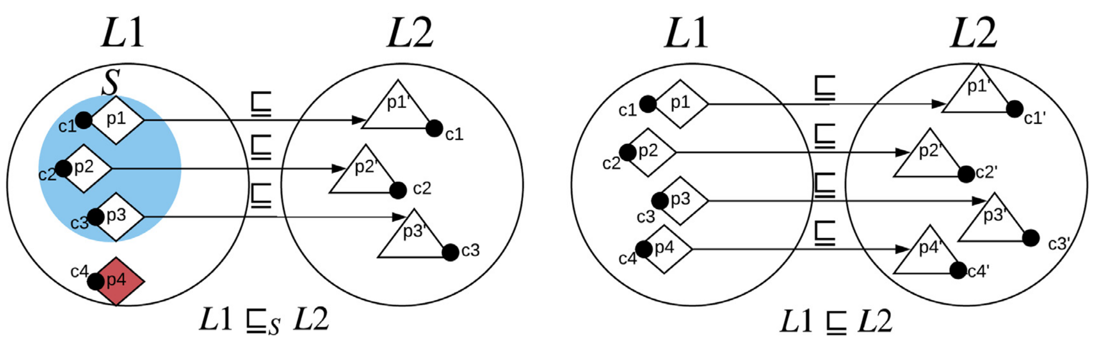

Fig. 2. Partial Refinement (left) versus Refinement (right).

$L \sqsubseteq_S L'$, whenever

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

To 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' with the resulting product line after removing *LEDS_RENESAS_TPU*. Thus, S would be the set of all configurations that do not contain *LEDS_RENESAS_TPU*.

Since the only modification is the feature removal, and we filter the respective changed products by ensuring refinement only for configurations in S, partial refinement holds. Partial refinement would not hold, for example, if S included configurations containing *LEDS_RENESAS_TPU*, as S would not be a subset of $[[F']]$. Hence, considering that we give guarantees that the other products are refined, developers would only need to test at most products that had *LEDS_RENESAS_TPU*. This could consequently increase productivity. The previous theory gives no guarantees for this case, so developers would have no support. We should notice that we are comparing products generated with the same c for the two product lines, so configurations cannot change. Therefore, *feature names matter*. For this reason, feature renaming is not a partial refinement. We revisit this topic later and present a more general notion of partial refinement to cover feature renaming.

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 4, 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 an arbitrary product line $L = (F, A, K)$, and a set of configurations S, if $S \subseteq [[F]]$, then $L \sqsubseteq_S L$.*

Proof. Let $L = (F, A, K)$ be an arbitrary product line. By Definition 4, we have to prove that $S \subseteq [[F]]$ and $\forall c \in S \cdot [[K]]_c^A \sqsubseteq [[K]]_c^A$. The first condition is already assumed by the theorem and the second follows from asset refinement reflexivity (Borba et al., 2012). □

One might want to consecutively perform partial refinement operations, and the transitivity property guarantees that this is feasible, and that it might result in refined products. Given that the consecutive partial refinement operations 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, as illustrated in Fig. 3, given a product line L_1, one could first perform a partial refinement operation, resulting in a product line L_2, and then perform another change, obtaining L_3. Assuming that S and T are the sets of configurations refined in each step, S would be the set of configurations $c1$, $c2$ and $c3$, and T would be the set of configurations $c1$ and $c2$. Assuming that S and T are different, the resulting product line L_3 does not partially refine L_1 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 L_3 partially refines L_1 for the configurations that are in both sets: $S \cap T$. This notion is formalised in Theorem 2.

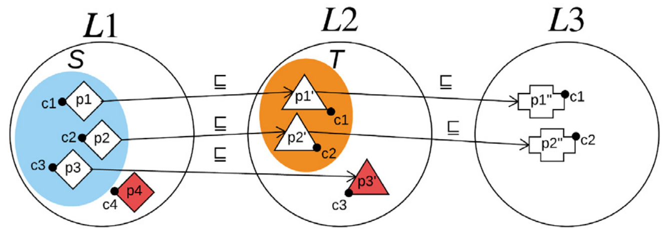

Fig. 3. Partial refinement transitivity.

Theorem 2 (Partial product line refinement transitivity)**.** *For arbitrary product lines L_1, L_2, L_3, and set of configurations S and T, if $L_1 \sqsubseteq_S L_2$ and $L_2 \sqsubseteq_T L_3$, then $L_1 \sqsubseteq_{S \cap T} L_3$.*

Proof. Let $L_1 = (F_1, A_1, K_1)$, $L_2 = (F_2, A_2, K_2)$ and $L_3 = (F_3, A_3, K_3)$ be arbitrary product lines. Assume that $L_1 \sqsubseteq_S L_2$ and $L_2 \sqsubseteq_T L_3$. By Definition 4, this amounts to

$$S \subseteq [[F_1]] \wedge S \subseteq [[F_2]] \tag{1}$$

$$\forall c \in S \cdot [[K_1]]_c^{A_1} \sqsubseteq [[K_2]]_c^{A_2} \tag{2}$$

$$T \subseteq [[F_2]] \wedge T \subseteq [[F_3]] \tag{3}$$

$$\forall c \in T \cdot [[K_2]]_c^{A_2} \sqsubseteq [[K_3]]_c^{A_3} \tag{4}$$

We then have to prove that

$$(S \cap T) \subseteq [[F_1]] \wedge (S \cap T) \subseteq [[F_3]] \tag{5}$$

and

$$\forall c \in S \cap T \cdot [[K_1]]_c^{A_1} \sqsubseteq [[K_3]]_c^{A_3} \tag{6}$$

We can prove Predicate (5) by using Predicate (1) and Predicate (3). To prove Predicate (6), assuming an arbitrary $c \in S \cap T$, we have to prove that $[[K_1]]_c^{A_1} \sqsubseteq [[K_3]]_c^{A_3}$. Properly instantiating c in Predicate (2) and Predicate (4), we have that $[[K_1]]_c^{A_1} \sqsubseteq [[K_2]]_c^{A_2}$ and $[[K_2]]_c^{A_2} \sqsubseteq [[K_3]]_c^{A_3}$. The proof then follows by asset set refinement transitivity (Borba et al., 2012). □

Note that S and T might be disjoint. For example, let us consider, for simplicity, a product line with two features A and B only. One could first remove feature A and then feature B. S in this case would be the set of product configurations that do not have A, that is the configuration that has just the feature B. In the second transformation, however, T corresponds to product configurations that do not have B, that is the product configuration that has A only. So, we would not be able to give guarantee for any product. This may naturally happen in a number of scenarios but for others we still give support after several transformations. This is important when considering a system such as the Linux Kernel, which has thousands of features. Therefore, we expect that for many of the possible evolution scenarios, one could still be supported after performing consecutive partially safe transformations.

We also provide a property that allows developers to choose a subset of S, given that partial refinement holds for S. This property is formalised in Theorem 3. So, if a product line refines another in terms of S, this is also true for any subset of S. We try to make S as large as possible to potentialize our support, but this does not prevent one from choosing a smaller subset.

Theorem 3 (Partial refinement holds for subset)**.** *For arbitrary product lines L and L', and sets of configurations S and S', if $L \sqsubseteq_S L'$ and $S' \subseteq S$, then $L \sqsubseteq_{S'} L'$.*

Proof. For arbitrary product lines $L = (F, A, K)$ and $L' = (F', A', K')$, and sets of configurations S and S', by assuming $L \sqsubseteq_S L'$ and $S' \subseteq S$, which amounts to

$$\forall c \in S \cdot [[K]]_c^A \sqsubseteq [[K']]_c^{A'} \tag{7}$$

and

$$\forall c \in S' \cdot c \in S \tag{8}$$

We then have to prove

$$\forall c \in S' \cdot [[K]]_c^A \sqsubseteq [[K']]_c^{A'} \tag{9}$$

For an arbitrary c in S', we have to prove that $[[K]]_c^A \sqsubseteq [[K']]_c^{A'}$. Properly instantiating c in Predicate (8), we have that $c \in S$. So, we can instantiate c in Predicate (7) and this concludes our proof. □

3.2. Compositionality

To simplify reasoning about partial refinement, it is important to derive compositionality properties from our definition. These are useful, for example, when the product line main elements evolve separately to be later integrated to generate products. In this context, one might need to change a specific artefact, for instance, the FM, without changing the AM and CK. In this case, instead of using the definition to verify partial refinement after changes are applied to a specific artefact, we could rely on a theorem and verify partial refinement in a simpler way. Developers could also modify different product line elements. We analyse these scenarios and whether such modifications preserve product line partial refinement. Compositionality theorems are provided in the existing refinement theory (Borba et al., 2012), so it would be important to provide the same kind of modular support for partial refinement too. So, instead of using the definition, one could use the compositionality theorems provided here.

3.2.1. FM partial equivalence

We first analyse the FM. Developers often desire to change feature types and dependencies. For example, a mandatory feature may become optional. The refinement theory already provides support for this and other FM refinement scenarios. According to the FM refinement notion (Borba et al., 2012), a FM refines another when the configurations of the initial FM are a subset of the evolved FM configurations. FM refinement often implies product line refinement and partial refinement (considering S to be all initial configurations), as we would be able to generate all existing products in the new product line. However, in scenarios such as a changing a feature from optional to mandatory, FM refinement is not applicable because we do not simply increase the set of configurations. In this case, the FMs may share configurations, but the new FM might have configurations that are extensions of the configurations of the initial FM. Thus, to provide support for such scenarios, we establish a partial FM equivalence notion, relating two FMs in terms of a common set of configurations S. This allows the initial FM to have configurations absent from the final FM. This contrasts with previous FM equivalence and refinement notions, that require the initial FM semantics to be equal or a subset of the final FM semantics (Borba et al., 2012).

Definition 5 (Feature model partial equivalence)**.** For arbitrary feature models F and F', and a set of configurations S, F is equivalent to F' modulo S, denoted by $F \cong_S F'$, whenever

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

Now, we would be able to support developers when transforming a feature from optional to mandatory. Partial equivalence holds, if S is the set of configurations in the initial FM that already had the changed feature plus those that do not have its parent. We should notice that FM equivalence and refinement lead to FM partial equivalence, but the opposite does not hold.

As captured in Theorem 4, FM partial equivalence leads to product line partial refinement. Given a product line L, one can modify the FM, by adding, removing or modifying features and dependencies, 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 (Borba et al., 2012) and we deal with arbitrary changes to F that result in F', 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. This is the reason for requiring well-formedness. Partial refinement holds because we are not checking products whose configurations are not in S. Moreover, the partial 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 we only check configurations from S.

Theorem 4 (Feature model partial 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 ($wf(L')$), then $L \sqsubseteq_S L'$.*

Proof. For an arbitrary product line $L = (F, A, K)$, a FM F' and a set of configurations S, assume that $F \cong_S F'$. By Definition 5, this amounts to:

$$\forall c \in S \cdot c \in [[F]] \wedge c \in [[F']] \tag{10}$$

By Definition 4 we then need to prove that

$$S \subseteq [[F]] \wedge S \subseteq [[F']] \tag{11}$$

and

$$\forall c \in S \cdot [[K]]_c^A \sqsubseteq [[K]]_c^A \tag{12}$$

and

$$wf(L') \tag{13}$$

We can prove Predicate (11) directly from Predicate (10), and Predicate (13) is assumed in the theorem. Finally, Predicate (12) is trivially true from asset set refinement reflexivitiy (Borba et al., 2012). □

3.2.2. AM partial refinement

Similarly to the FM, the AM may also be modified separately. Previous work shows that the source code is more frequently modified than the FM and CK (Dintzner et al., 2014). In these cases, one not necessarily modifies the FM and the CK. The existing AM refinement compositionality notion (Borba et al., 2012) is helpful only when all assets from the initial AM are refined by the evolved ones. But, for example, that is not true in bug fix scenarios, when changes are not safe in at least one asset.

According to Definition 6, the AMs must have the same domain. Additionally, for every asset a found in the initial AM A, there needs to be an asset a' in the evolved AM A' with the same name (an), that refines the initial one ($a \sqsubseteq a'$). AM refinement holds in several situations, like in a function renaming scenario. If we rename a function in each initial asset, all of them are refined by the new ones.

Definition 6 (Asset mapping refinement)**.** For asset mappings A and A', A is refined by A' whenever

$$(dom(A) = dom(A') \wedge$$
$$(\forall an \in dom(A) \cdot$$
$$\exists a, a' \cdot (an, a) \in A \wedge (an, a') \in A' \wedge \ a \sqsubseteq a'))$$

To support such scenarios, we define partial AM refinement. As stated in Definition 7, an AM partially refines another for a subset of names when refinement holds for the sub mappings derived from this subset. More specifically, the AM resultant from filtering the original AM A according to a set of asset names ns (that is formalised as $A \lhd ns$, which expands to $\{(n\colon Name,\ a\colon Asset) | (n, a) \in A \wedge n \in ns\}$) needs to be refined (according to Definition 6) by the AM obtained by filtering the new AM A' according to ns. In the case of a bug fix scenario, the new AM would partially refine the original one modulo the set of names of the assets not changed by the fix.

Definition 7 (AM Partial Refinement)**.** For arbitrary asset mappings A and A', and a set of asset names ns, A' partially refines A modulo ns, denoted by $A \sqsubseteq_{ns} A'$, whenever

$$(A \lhd ns) \sqsubseteq (A' \lhd ns),$$

CK Evaluation

AM partial refinement implies product line partial refinement, since products containing only asset names in ns are not affected. As a product is represented as a set of assets, we need to discover which products (and their configurations) have assets whose names are in ns to precisely express the set of refined products after a change in an AM. Thus, in this context, we assume the CK semantics function can be decomposed in terms of an evaluation function $(\!|\_|\!)$. We use $(\!|K|\!)^A_c$ to denote a call to the evaluation with CK K, AM A and configuration c. This function is similar to the CK semantics function, but instead of returning a set of assets, it returns assets and their names in the form of an AM (the submapping of the original AM containing only the assets used to build the product). This way, we are able to maintain the mapping and check if a product has an asset associated with a specific name in the AM. We use the term assumption because we are not actually providing the function body or implementation, but only defining its signature. Due to this, the properties are also assumptions, which is why we have them as axioms, since we are not able to prove them, given that we do not have a concrete function definition.

Assumption 1 (Configuration knowledge evaluation)**.**

$$(\!|\_|\!) : CK \to AM \to Configuration \to AM$$

To illustrate how the evaluation function works, consider the product line example shown in Fig. 1 and assume that F, A and K correspond to its FM, AM and CK, respectively. This product line has at least two features: *LEDS_RENESAS_TPU* and *LEDS_CLASS*. If we have a configuration $c = LEDS\_CLASS$, by calling the CK semantics function $[[K]]^A_c$, we would obtain the product containing only this feature, which would contain only the $leds - class.c$ asset. A call to the evaluation function $(\!|K|\!)^A_c$, however, would return the AM containing the information related to the features present in the configuration c, which would give us a single mapping containing $leds - class.c$ and its content. Note that they return similar results. The difference is that the evaluation function returns an AM and the semantics function returns only the respective assets. Thus, the CK evaluation function is also useful for establishing a correspondence between configurations and asset names. Given the semantics function, we only know the assets generated from a configuration, but we do not have the tracking of the asset names.

Here we do not concretely define the CK semantics function. We just define it in terms of the evaluation function, which is not defined because we do not deal with any particular CK notion in the general level. The evaluation function has is just given signature and satisfies a number of axioms. In our appendix we provide two types of CK semantics: (1) the compositional one, which corresponds essentially to the union of assets mapped to the selected features, and (2) the transformational one, which does not simply "joins" assets, but also applies transformations. In (1), we cannot select part of an implementation asset for a determined feature. We can think of (2) as being the more general case where we have a block of code denoting the scope of a feature, instead of an entire implementation asset. The semantics function is simply the image of the AM returned by the evaluation function. As we want to keep our theory language independent, we do not offer a concrete definition because that would be inevitably specific to asset, class and FM languages. The result of the semantics function is just collecting the assets in the resulting AM from the evaluation function, and ignoring their names. The assets of an AM constitute its image, which is denoted by $A\langle\_\rangle$ for an AM A. The semantics function is then defined as $img((\!|K|\!)^A_c)$, which means the image of $(\!|K|\!)^A_c$. We are assuming that the evaluation function captures the other steps in the product generation process.

Definition 8 (Configuration Knowledge Semantics)**.** Let (F, A, K) be a product line and c be a configuration. Then, $[[K]]^A_c = img((\!|K|\!)^A_c)$

Sanity Conditions

To make sure the CK evaluation function for different languages captures the essence of the product generation process, we rely on axioms that capture sanity conditions that such a function should satisfy. These axioms rule out abnormalities such as changing an asset name during the product generation process. First, we should guarantee that the resulting asset names from the generated products belong to the original asset mapping of the product line. Otherwise, we could have dangling assets, which does not make sense. For this reason, we formalise this in Axiom 1.

Axiom 1 (CK evaluation must preserve AM domain)**.** For arbitrary AM A, CK K, configuration c, $dom((\!|K|\!)^A_c) \subseteq dom(A)$

We also state that unused assets do not influence the CK semantics in Axiom 2. If the asset names of a determined product, which is the result of applying the CK semantics function, belong to a set of names ns, the result of the CK evaluation should be the same using the entire AM or the filtered AM according to ns. The reasoning is that if the other names are not present in the respective product, they can be discarded.

Axiom 2 (Unused assets do not influence CK semantics)**.** For an arbitrary AM A, a CK K, a set of asset names ns and a configuration c, if $dom((\!|K|\!)^A_c) \subseteq ns$, then $[[K]]^A_c = [[K]]^{A \lhd ns}_c$

The third constraint is established in Axiom 3. The evaluation function not only is forbidden to create new asset names, but for asset mappings with equal domain, the resulting domain after applying the evaluation function also needs to be equal. So, the evaluation function preserves equality over the domain of AM.

Axiom 3 (Evaluation preserves equality over AM domain)**.** For arbitrary AMs A and A', CK K, and configuration c, if $dom(A) = dom(A')$, then $dom((\!|K|\!)_c^A) = dom((\!|K|\!)_c^{A'})$.

The introduced axioms are essential to avoid an arbitrary evaluation function, and they do not overly restrict the applicability of our theory, since they are mostly well-formedness conditions, to prevent abnormal situations. To guarantee that both compositional and transformational CKs satisfy these axioms, we instantiate this theory using both CK notions and prove them. The axioms are also important for stating and proving the AM compositionality theorem. We also found that all product lines from the evolution scenarios analysed, which are detailed in Section 6, obey the three axioms. This confirms our intuition that they are reasonable to be assumed.

AM Partial Refinement Compositionality

We finally establish Theorem 5, which states that partial AM refinement implies partial product line refinement. We calculate the set of configurations S for these situations based on the AM commonalities and differences. Configurations from S must not generate products containing the names that are not in ns, since as already discussed earlier in this section, these products are not refined. So, partial refinement does not hold for them. Alternatively, configurations that lead to products containing assets in the scope of ns are refined. To define S, we use a restriction operator $\upharpoonright$ that takes the three elements of a product line (F, A, K) and a finite set of asset names ns. It then yields configurations whose products have only assets that are in ns, as can be seen in Definition 9. However, it is not enough to filter configurations considering the original AM, since A and A' have different domains. So, we need to define S as the intersection of filtering both AMs according to ns (which is given by $((F, A, K)\upharpoonright ns)\cap((F, A', K)\upharpoonright ns)$).

Definition 9 (Filtering Configurations by Asset Names)**.** Let (F, A, K) be a product line and ns be a set of asset names. Then, $(F, A, K) \upharpoonright ns = \{c : Conf \mid c \in [[F]] \wedge dom((\!|K|\!)_c^A) \subseteq ns\}$

Theorem 5 (Asset mapping partial refinement compositionality)**.** *For product lines $L = (F, A, K)$ and $L' = (F, A', K)$, and a finite set of asset names ns, if $A \sqsubseteq_{ns} A'$ then $L \sqsubseteq_S L'$, where $S = (F, A, K) \upharpoonright ns \cap (F, A', K) \upharpoonright ns$.*

Proof. For an arbitrary PL $L = (F, A, K)$, an AM A' and a finite set of asset names ns. We have to prove that $L \sqsubseteq_S L'$, where $L' = (F, A', K)$ and $S = ((F, A, K) \upharpoonright ns) \cap ((F, A', K) \upharpoonright ns)$. According to Definition 4, $L \sqsubseteq_S L'$ expands to

$$S \subseteq [[F]] \wedge \forall c \in S \cdot [[K]]_c^A \sqsubseteq [[K]]_c^{A'} \tag{14}$$

It is true that $S \subseteq [[F]]$, since this is expressed in the definition of S and of the operator. Then, we need to prove that, for an arbitrary c in S, $[[K]]_c^A \sqsubseteq [[K]]_c^{A'}$. By properly instantiating K, A, ns, and c in Axiom 2, we have that $[[K]]_c^A = [[K]]_c^{A \lhd ns}$. The condition $dom((\!|K|\!)_c^A) \subseteq ns$ is satisfied due to S definition. Using Axiom 2 again, properly instantiated with K, A', ns, and c, we also have $[[K]]_c^{A'} = [[K]]_c^{A' \lhd ns}$. By replacing this in Predicate (14), we then need to prove that $[[K]]_c^{A \lhd ns} \sqsubseteq [[K]]_c^{A' \lhd ns}$. Using Definition 7, we have that $(A \lhd ns) \sqsubseteq (A' \lhd ns)$.

Since asset mapping refinement implies product line refinement (Borba et al., 2012), we have that $\forall am1, am2 \cdot am1 \sqsubseteq am2 \Rightarrow \forall K, c \cdot wf([[K]]_c^{am1}) \Rightarrow wf([[K]]_c^{am2}) \wedge [[K]]_c^{am1} \sqsubseteq [[K]]_c^{am2}$. Instantiating this equation with $am1 = A \lhd ns$ and $am2 = A' \lhd ns$, the first condition holds because (F, A, K) is a product line, and by definition, every product line is well-formed. So, this is enough to prove that $[[K]]_c^{A \lhd ns} \sqsubseteq [[K]]_c^{A' \lhd ns}$. □

3.2.3. CK partial equivalence

Now, we analyse scenarios where the CK structure is changed in isolation and how this impacts the entire product line. Developers may need to modify the CK only, and it is important to support them in these situations not only with our definition of partial refinement, but also with a CK partial equivalence notion. An example of such scenario would be adding a mapping between existing features and artefacts, assuming a compositional CK. In this case, the FM and the AM do not suffer any change. Only the CK is modified. Moreover, this scenario would not be considered CK refinement because features from this new mapping may suffer behavioural change, thus making certain configurations not to be refined.

To address this and other evolution scenarios, we formalise a partial equivalence notion to represent partial refinement changes regarding the CK only. Notions to deal with refinement scenarios have already been proposed (Borba et al., 2012). However, several of the possible changes involving the CK are not refinements. Definition 10 generalises all possible safe changes to the CK. We state that for a set of configurations S, the products generated using the original and final CKs are equal. If the CKs are equal, S could be the semantics of the FM, that is, the set of all valid configurations. In contrast, S could eventually be empty, and it would not be possible to provide any kind of support or guarantee.

Definition 10 (Configuration knowledge partial equivalence)**.** For arbitrary CKs K and K', and a set of configurations S, K is equivalent to K' modulo S, denoted by $K \cong_S K'$, whenever

$$\forall am, c \in S \cdot [[K]]_c^{am} = [[K']]_c^{am}$$

Configuration Knowledge partial equivalence implies product line partial refinement. This is shown in Theorem 6. Considering an arbitrary product line L, let us suppose that a change is made to the CK of L but preserving the set of configurations S. We then obtain L', and we can say that L' partially refines L according to S as long as S is a subset of the valid configurations generated from the FM of L. Just to give an example, considering that we obtain K' by removing a mapping from a hypothetical feature F to an asset a present in K. In this case, S would be the set of configurations in the FM that do not have F. Products containing F might not be refined, since they will not have the asset a as before. So, configurations containing F cannot be included in S. Since K is equivalent to K' according to S and S is a subset of F configurations, L' refines L according to the same S.

Theorem 6 (Configuration knowledge partial equivalence compositionality)**.** *For a product line $L = (F, A, K)$, a CK K', and a set of configurations S, let $L' = (F, A, K')$. If $K \cong_S K'$, $S \subseteq [[F]]$ and L' is well-formed, then $L \sqsubseteq_S L'$.*

Proof. For an arbitrary product line $L = (F, A, K)$, a CK K' and a set of configurations S, assume that $K \cong_S K'$ and $S \subseteq [[F]]$. This amounts to:

$$\forall am, c \in S \cdot [[K]]_c^{am} = [[K']]_c^{am} \tag{15}$$

By Definition 4 we then need to prove that

$$S \subseteq [[F]] \tag{16}$$

and

$$\forall c \in S \cdot [[K]]_c^A \sqsubseteq [[K']]_c^A \tag{17}$$

We are already assuming Predicate (16). So, for an arbitrary c in S, we need to prove that $[[K]]_c^A \sqsubseteq [[K']]_c^A$. By properly instantiating am and c in Predicate (15) with A and c, we have $[[K]]_c^A = [[K']]_c^A$. So, we can replace $[[K']]_c^A$ by $[[K]]_c^A$ and the proof follows from asset set refinement reflexivitiy (Borba et al., 2012). □

3.3. Combining different refinement and partial refinement notions

We also reason about compositionality in terms of combining different refinement notions, since the refinement and partial refinement theories are complementary. Thus, practitioners may desire to interleave refinement and partial refinement operations. For improvements or adding new features with behaviour preservation, one can use the refinement theory. After that, developers may need to remove a feature, such as the removal scenario illustrated in Section 2. For such cases, the partial refinement notion should be used. 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 a subset of products.

Refinement and partial refinement

When a partial refinement over S is followed by a refinement, we would ideally have partial refinement for products in S by transitivity. However this does not hold because, in the refinement transformation, feature names do not matter, contrasting with the partial refinement notion. In fact, as Definition 3 admits configurations to change, refinement is not necessarily a particular case of partial refinement even when S is equal to the set of all valid configurations.

To support interleaving of safe and partially safe changes, Definition 11 describes a more general partial refinement notion that allows configurations to change according to a renaming function f. The function f maps configurations from the initial to the final feature models. Then, given an initial configuration c from the initial feature model, 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 $f(c) = c[P'/P]$. This function takes a configuration c and returns c if c does not have the feature P. Otherwise, it gives a new configuration c' as result, that is equal to c, except that every occurrence of P is replaced by P'.

Definition 11 (Weak partial refinement)**.** For arbitrary product lines $L = (F, A, K)$, $L' = (F', A', K')$ and a function f: $[[F]] \rightarrow [[F']]$, L' weakly partially refines L modulo f, denoted by $L \sqsubseteq_f L'$, whenever

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

The partial refinement notion is a particular case of Definition 11 (when f is the identity function over S). Thus, this weaker notion supports situations where configurations change, which are not covered by the default partial product line refinement notion (Definition 4). Since the weak definition is more general, we could have it instead of having both partial refinement relations. However, Definition 4 is less complex to reason about, and it covers the majority of scenarios, unless developers need to deal with feature renaming, so we decided to keep both.

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 refined configurations were the same as the ones refined in the first step. Hence, even assuming two refinement operations in terms of the same S, transitivity does not hold for S.

Similarly to Definition 3, the weak partial refinement relation is also a preorder, as we should support developers in stepwise refinement. In Theorems 7 and 8, we formalise the reflexivity and transitivity properties. A product line partially refines itself, according to Definition 11, when the function f is an identity. Otherwise, it makes no sense to compare different products in the same product line.

Theorem 7 (Weak partial refinement reflexivity)**.** *For an arbitrary product line $L = (F, A, K)$, and a function f: Conf $\rightarrow$ Conf, if f is the identity function and $dom(f) \subseteq [[F]]$, then $L \sqsubseteq_f L$.*

Proof. Let $L = (F, A, K)$ be an arbitrary product line. By Definition 11, we have to prove that, for an arbitrary c in $dom(f)$, $[[K]]_c^A \sqsubseteq [[K]]_{f(c)}^A$. Since f is the identity function, we can replace $f(c)$ by c and the proof follows from asset refinement reflexivity (Borba et al., 2012). □

For the transitivity property, the reasoning is similar to Theorem 2. Instead of giving refinement guarantees for the intersection of the two sets of configurations, we compose the two functions defined for each evolution step.

Theorem 8 (Weak partial refinement transitivity)**.** *For arbitrary product lines L_1, L_2, L_3, and functions f: Conf $\rightarrow$ Conf and g: Conf $\rightarrow$ Conf, if $L_1 \sqsubseteq_f L_2$ and $L_2 \sqsubseteq_g L_3$, then $L_1 \sqsubseteq_{g \circ f} L_3$.*

Proof. Let $L_1 = (F_1, A_1, K_1)$, $L_2 = (F_2, A_2, K_2)$ and $L_3 = (F_3, A_3, K_3)$ be arbitrary product lines. Assume that $L_1 \sqsubseteq_f L_2 \wedge L_2 \sqsubseteq_g L_3$. By Definition 11, this amounts to:

$$\forall c \in dom(f) \cdot [[K_1]]_c^{A_1} \sqsubseteq [[K_2]]_{f(c)}^{A_2} \tag{18}$$

$$\forall c \in dom(g) \cdot [[K_2]]_c^{A_2} \sqsubseteq [[K_3]]_{g(c)}^{A_3} \tag{19}$$

We then have to prove that

$$\forall c \in dom(g \circ f) \cdot [[K_1]]_c^{A_1} \sqsubseteq [[K_3]]_{g(f(c))}^{A_3} \tag{20}$$

For an arbitrary $c \in dom(g \circ f)$, we need to prove that $[[K_1]]_c^{A_1} \sqsubseteq [[K_3]]_{g(f(c))}^{A_3}$. Since the domain of $g \circ f$ is equal to the domain of f, we can instantiate c in Predicate (18). We then have $[[K_1]]_c^{A_1} \sqsubseteq [[K_2]]_{f(c)}^{A_2}$. Properly instantiating c in Predicate (19) with $f(c)$, we then have $[[K_2]]_f^{A_2}(c) \sqsubseteq [[K_3]]_{g(f(c))}^{A_3}$. The proof then follows by asset set refinement transitivity (Borba et al., 2012). □

When one applies a partial refinement followed by a refinement, we have a weak partial refinement. A possible scenario of such situation is when one changes an asset in a non behaviour-preserving way and then renames a feature,. 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. Suppose that feature P was renamed to P', 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 formalised in Theorem 9. When partial refinement is followed by refinement, there is a function that maps configurations from S to the final product line, so that weaker partial refinement holds.

Theorem 9 (Partial refinement and refinement)**.** *For product lines L_1, L_2 and L_3 and a set of configurations S, let F_3 be the FM of L_3. If $L_1 \sqsubseteq_S L_2$ and $L_2 \sqsubseteq L_3$, then, for some function f: $S \rightarrow [[F_3]]$, $L_1 \sqsubseteq_f L_3$.*

When a refinement (Definition 3) occurs, we can derive a function that maps configurations. Given an initial configuration from the initial FM, the function arbitrarily chooses a configuration from the final FM so that product refinement holds. So, in this case we could say that there is a function g: $[[F_2]] \rightarrow [[F_3]]$ that maps configurations from L_2 to L_3. In the first case, when we have a partial refinement (Definition 4), we require that configurations do not change, differently from Definition 11. So, we can rely on the identity function I: $S \rightarrow [[F_2]]$, since the initial configuration is equal to the final one. Thus, f: $S \rightarrow [[F_3]]$ in Theorem 9 would be the composition of g with the identity function I. Product refinement then holds by transitivity.

If the operations are conducted in the opposite order (refinement followed by partial refinement), the reasoning and end result are analogous, so we omit the details here. The respective theorem and proof can be found in our online appendix.

Name aware refinement and partial refinement

The composition of refinement and partial refinement is intricate, because refinement allow changing feature names and configurations. A plainer composition can be established with a name aware, stronger notion of refinement. It is stronger than the standard notion in the sense that it gives less flexibility in terms of configurations. This definition supports less scenarios when compared to Definition 3. Feature renaming, for instance, is not a refinement according to this notion. Since configurations are usually sets of feature names, when changing such names, configurations containing them are impacted.

Definition 12 (Name aware 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

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

Previous work has shown that this relation has similar properties to the refinement relation, like being a preorder (Borba et al., 2012). This notion (Definition 12) is similar to the partial refinement notion (Definition 4) in the sense that it does not allow any change in configurations. For product lines L and L', name aware refinement implies 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 name aware refinement, we have a partial refinement, as shown in Theorem 10. If the refinements are performed in the opposite order, the result is also a partial refinement.

Theorem 10 (Partial and name aware refinement)**.** *For product lines L_1, L_2 and L_3 and set of configurations S, if $L_1 \sqsubseteq_S L_2$ and $L_2 \preceq L_3$, then $L_1 \sqsubseteq_S L_3$.*

Proof. For arbitrary product lines $L_1 = (F_1, A_1, K_1)$, $L_2 = (F_2, A_2, K_2)$ and $L_3 = (F_3, A_3, K_3)$ and set of configurations S, we assume $L_1 \sqsubseteq_S L_2$ and $L_2 \preceq L_3$. This expands to

$$S \subseteq [[F_1]] \wedge S \subseteq [[F_2]] \tag{21}$$

$$\forall c \in S \cdot [[K_1]]_c^{A_1} \sqsubseteq [[K_2]]_c^{A_2} \tag{22}$$

and

$$\forall c \in [[F_2]] \cdot c \in [[F_3]] \wedge [[K_2]]_c^{A_2} \sqsubseteq [[K_3]]_c^{A_3} \tag{23}$$

We then have to prove

$$S \subset [[F_1]] \wedge S \subset [[F_3]] \tag{24}$$

and

$$\forall c \in S \cdot [[K_1]]_c^{A_1} \sqsubseteq [[K_3]]_c^{A_3} \tag{25}$$

Predicate (24) is true from Predicate (21) and Predicate (23). To prove Predicate (25), we assume an arbitrary c in S. We then properly instantiate c in Predicate (22) and Predicate (23), as any c in S is also in $[[F_2]]$ vide Predicate (21) and the proof follows by asset set refinement transitivity Borba et al. (2012). □

$$\begin{tikzcd}
L_1 \arrow[r, "\sqsubseteq_S"] \arrow[d, "\preceq"'] & L_2 \arrow[d, "\preceq"] \\
L_3 \arrow[r, "\sqsubseteq_S"'] & L_4
\end{tikzcd}$$

Fig. 4. Commutative diagram (refinement and partial refinement).

Commutativity of name aware refinement and partial refinement

Finally, we show that name aware refinement and partial refinement transformations lead to the same product line when applied in different orders. For instance, given a product line L_1, suppose that a developer performs a name aware refinement, such as locally refactoring an asset, obtaining L_3, and then partially refines the product line by removing a feature, obtaining L_4. Fig. 4 represents a commutative diagram that shows that if we instead first apply this same partial refinement operation (yielding L_2) and then refine the asset, we obtain the same L_4. Thus, in this case, the order in which the transformations are applied does not matter.

Properties like this 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 can produce the same result. We formally derive and prove two theorems from the commutative diagram structure shown in Fig. 4. We only present the proof for the first theorem; the other follows the same strategy. Both proofs can be found in our online appendix (Partial refinement theory website).

In Theorem 11, we give support in case developers are doing first a partial refinement and then a name aware refinement. The theorem establishes that there is an alternative way to obtain the same resulting product line, by performing the corresponding operations in the opposite order. Theorem 12 is analogous. This theorem has an extra condition when compared to the first one. This property only holds if S is a subset of the valid configurations generated by the initial product line L_1. 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 11 (Partial refinement and name aware refinement commute (1))**.** *For product lines L_1, L_2 and L_4, and a set of configurations S, if $L_1 \sqsubseteq_S L_2$ and $L_2 \preceq L_4$, then, for some product line L_3, we have $L_1 \preceq L_3 \wedge L_3 \sqsubseteq_S L_4$.*

Proof. For arbitrary product lines L_1, L_2, L_4 and a set of configurations S, we assume $L_1 \sqsubseteq_S L_2$ and $L_2 \preceq L_4$ to prove that

$$\exists L_3 \cdot L_1 \preceq L_3 \wedge L_3 \sqsubseteq_S L_4 \tag{26}$$

Instantiating L_3 with L_1 in Predicate (26), we then have to prove that $L_1 \preceq L_1$ and $L_1 \sqsubseteq_S L_4$. Since $\preceq$ is a preorder, $L_1 \preceq L_1$ trivially holds. To prove that $L_1 \sqsubseteq_S L_4$, we use partial refinement transitivity (Theorem 2). We just need to show that $L_2 \sqsubseteq_S L_4$. Since we assume that $L_2 \preceq L_4$, and we know that the name aware refinement is a specific case of partial refinement (when S is the entire set of valid configurations), this concludes the proof. □

Theorem 12 (Partial refinement and name aware refinement commute (2))**.** *For product lines L_1, L_3, L_4 and a set of configurations S. Let F_1 be the FM of L_1. If $S \subseteq [[F_1]]$, $L_1 \preceq L_3$ and $L_3 \sqsubseteq_S L_4$, then, for some product line L_2, we have $L_1 \sqsubseteq_S L_2 \wedge L_2 \preceq L_4$.*

3.4. Discussion

In the previous section, we establish a relationship between the partial refinement and name aware refinement relations, through a commutative diagram illustrated in Fig. 4. We believe that the diagram also holds if we replace $\preceq$ by $\sqsubseteq$. Nevertheless, to prove the correspondent theorems, we would need to enrich our theory. Instead of having only relations connecting product lines, we would also need transformation operations expressing how product lines change. Instead of considering just sets of product lines, our encoding would have also to consider sets of transformations or changes among product lines. The proof would then be made by induction over the set of possible transformations.

4. Partially safe evolution templates

As mentioned in Section 3, the partial refinement theory can be applied to different contexts than the refinement theory (possibly even more). In this section, we illustrate such contexts and define templates that are abstractions of recurrent practical evolution scenarios. We defined such templates based on preexisting refinement templates (Neves et al., 2015; Borba et al., 2012), by changing conditions to allow partially safe changes. Our templates help because they provide guidance on how to evolve a product line guaranteeing safe evolution for a subset of the products. Moreover, developers do not need to reason over the partial refinement definition for these scenarios; the templates already provide some guidance. Templates might also avoid errors during the evolution process and increase developers confidence, since they provide guidance on how to change a product line.

A template has a left-hand side pattern (LHS) and a right-hand side pattern (RHS), stating syntactic and semantic conditions for the transformation to be applied. They correspond to abstractions that capture properties of the initial and evolved product lines, respectively. We make use of meta-variables to represent the initial and evolved product line elements. An element is supposed to be unchanged when the corresponding meta-variable is present in both sides. In case one follows the syntactic and semantic rules established by templates, partial refinement holds for a specified subset of products S.

We represent the initial and evolved product lines with the three elements: FM, AM and CK and we show them in detail when they are changed. We use a tree notation to represent the FM. Although the FM structure can be large, with several features, we choose to show only the ones affected in the particular evolution scenario. So, if the FM in the template contain only feature F, it is implicit that there might be upper and sub-trees attached. Although we do not show in the templates, we assume that the FM contains cross-tree constraints, which are formulae used to build implications involving features usually not directly related in the tree. For example, features P and O can be siblings and one would have the constraint $P \Rightarrow O$ to guarantee that whenever P is selected, O is also selected. As we define in Section 3, the AM is basically a set of pairs, each pair containing an asset name and an asset. We show these pairs between braces, and each pair has the form of $n \mapsto a$, meaning that the asset name n is associated with the asset a. The CK is represented as a table-like structure with two columns: on the left-side column we have feature expressions containing features names and first-order logic boolean operators. These expressions could be, for instance, $P \wedge O$ (the case where both features P and O are selected) or $\neg P \vee O$ (the case where we have either not P or O). The right-side column depends on the CK language adopted. It can be either sets of asset names or transformations. We show each case later in detail.

The value of S is defined in terms of the FM, AM and CK of the product lines in the templates. Establishing S this way helps to understand the change impact, since products in the scope of S are not impacted. We should remember that product line refinement holds for any configuration in S (according to Theorem 3 from Section 3), so one might choose to work with a smaller subset of S. We assume two notations for the CK structure: compositional and transformational (see Neves et al., 2015). So we provide three sets of templates, one in each subsection, first focusing on templates for the compositional notation, then on templates for the transformational notation, and finally on more general templates that apply to both notations.

4.1. Compositional templates

In this section, we present templates that use the compositional CK notation, consisting of a table-like structure with two columns. The left column has feature expressions (enabling conditions) and the right column has asset names, indicating that a given configuration yields a product containing the names in the right whenever the expression in the left evaluates to true. In the following, we analyse a number of possible scenarios of partially safe evolution.

Remove Feature

We first analyse feature removal situations, which is an usual scenario in a product line development context. One often decides to exclude features for diverse reasons (Passos et al., 2015); for instance, they are no longer used or not needed by customers. We define the REMOVE FEATURE template in Fig. 5. Products that did not have the removed feature in the original product line keep the same behaviour, and the others might not be refined. We show the three elements before and after the feature removal in Fig. 5. The feature model, which is shown in a tree-like notation, the asset mapping, which is a set of mappings from asset names to assets (inside curly brackets), and the configuration knowledge, which is shown in a table-like structure. In this template, the three product line elements are changed.

By syntactically analysing the REMOVE FEATURE template in Fig. 5, we observe that the initial FM (F), has the O feature, which is removed, and consequently, the resulting FM (F') does not have it. We also notice that O is P's child. Nothing else is changed in the FM, which might have other features beyond O and P. We assume that the initial CK has references to O, so from the LHS to the RHS, every row in the CK (like the one containing e' and n') referencing O is removed. If the CK has no references to O, the feature could be removed directly but this scenario would actually consist in a product line refinement. The AM has names such as n and n' mapped to a and a'. Similarly to the FM and CK, the AM also loses a set of mappings, like a' which implements O.

The guarantees provided by the template only hold if some conditions are valid. We need to make sure that when e' is true, O

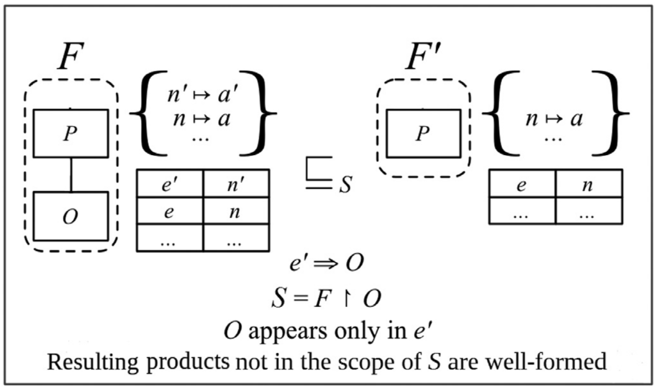

Fig. 5. REMOVE FEATURE compositional partial refinement template.

has been selected, otherwise it would make no sense to exclude this expression 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. We could use O instead of e', but this would restrict the template applicability. When using e', we are allowing any expression where O is true. When a feature is removed, the intuition is that the products that did not have the respective feature do not change, so behaviour is preserved. The other ones might not be compatible to any product in the new product line because they lose functionality, unless a' adds no extra behaviour 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 $\wedge$, $\neg$ and $\vee$. The expression $P \wedge Q$, for instance, is satisfied by a configuration c when c has the P and Q features. For an arbitrary FM F and a feature expression e, we use $F \upharpoonright e$ to denote the set of configurations in $[[F]]$ that do not satisfy e. This is formalised in Definition 13. 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 is to avoid that another expression references O either directly or indirectly. Finally, we also need a well-formedness condition to guarantee that the products not refined (the ones that had O in the initial product line) remain well-formed. Since we assume that assets are removed from the initial to the evolved product line, we cannot guarantee that existing products remain well-formed, except those in S.

Definition 13 (Filtering Configurations by Feature Expression)**.** Let F be a FM and e be a feature expression. Then, $F \upharpoonright e = \{c : Conf \mid c \in [[F]] \wedge \neg sat(c, e)\}$

The Remove feature template does not assume that O is a leaf feature. However, when O is removed, the subtree under O is also removed according to the template. One might need to remove O, but keeping its children. Thus, we would need another template, which would be a variation of the template illustrated in Fig. 5, to deal with such scenario and this is part of our future work.

Strictly, this template does not match the example discussed in Section 2, but it is compatible with a slight variation of the template, where two assets are removed. 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 except for this mapping. The Linux AM could be represented by mappings between the file 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. Assuming that the resulting product line is well-formed, all conditions are satisfied. S is $F \upharpoonright$ *LEDS_RENESAS_TPU*. Thus, refinement holds for these configurations. The other products are not refined since they have the removed feature, thus not preserving behaviour. Differently from product line refinement (Definition 3), which requires every product in the initial product line to be compatible with at least one product in the new product line, partial refinement requires refinement for a subset of the initial products. Therefore, in this case, only products without *LEDS_RENESAS_TPU* are refined.

4.2. Transformational templates

As shown in the previous section, feature removal is a possible partially safe evolution scenario developers might face in practice. We also have templates to deal with other scenarios, such as asset additions, removals and changes to the CK (Sampaio et al., 2016). Here, we deal with the same scenarios, but assuming that the CK may also have transformations. Our theory covers two types of transformations: *preprocess* and *select*. By using the former, one can limit the scope of a feature by using *#ifdefs* around the corresponding code block. Thus, the included assets might not be the original ones; they may suffer changes during the compilation process. The latter simply selects the corresponding asset, so it does not change the asset itself.

Remove Feature

Developers may need to remove features for diverse reasons, as already explained in Section 4.1. The compositional Remove feature template assumes that the CK has feature expressions and asset names, whereas the transformational one deals with transformations instead of names. They also differ in the structures underlying their similar CK syntax. While the former definition consists of a set of items, the latter consists of a list. It is not feasible to work with these different representations at the same time. Additionally, a number of conditions may vary and impose restrictions on the artefacts format. For instance, one could assume an artefact with an *ifdef* block, while another simply does not deal with such structure. For these reasons, we present two versions of the Remove feature and other templates and this increases expressiveness of the partial refinement notion, as we are allowing different SPL languages. Thus, we also have a template to deal with feature removals but allowing the use of transformations in the CK and *ifdefs* in the AM structure. In Fig. 6, the three parts of the product line are affected. Similarly to the feature removal template for compositional CKs, we give support for the products that do not have the removed feature. Therefore, S is defined in the same way.

The O feature is removed from the initial FM F. Similarly to its compositional version, assets implementing the O feature should be removed from the product line. In this case, since we are able to transform assets, we can consider the use of preprocessing directives to implement features. So, instead of a single file implementing the entire O feature, we have a code snippet c inside the a asset where O is implemented. In the CK, the x tag activates the c code that implements O. Thus, in the new AM, this tag is not present, neither is the c code. Two lines are removed from the initial CK. Both have an expression e, which, if true, implies the presence of O. For this reason, both lines should be removed, otherwise we could have an ill-formed CK, that refers to features that do not exist anymore. The first transformation in the first line is *tag x*, which activates this tag. The transformation *preprocess n* generates a new asset considering activated tags. As a consequence, since the x tag is previously activated, c is included. If x had not been activated, c would not be included. There are three additional conditions in the template illustrated in Fig. 6. The first one is to make sure that O only appears in e. This is essential to guarantee that the remaining expressions in the resulting CK will not refer to a feature that was already removed. We also require the x tag to not appear in other CK lines. As this tag activates the code related to the removed feature O, it should also be removed. The first CK line refers to x, and it is also removed. Since we are assuming that x refers to the removed feature, it makes no sense to allow that other CK lines refer to x. This could imply in a ill-formed product line because after removing O, there would still be a tag referring

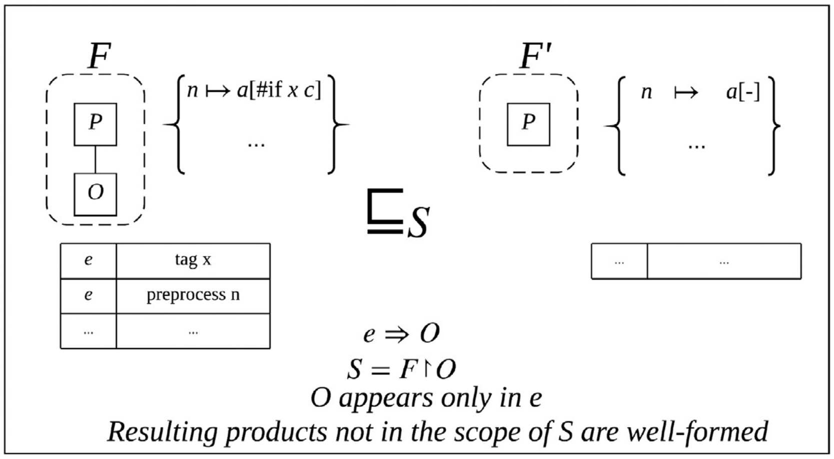

Fig. 6. Remove feature partial refinement transformational template.

to O. Finally, we need a well-formedness condition. We do not have control over the products that are not in the scope of S, since these products were affected by the feature removal. Consequently, we do not know if these products would still compile successfully, for example. We then need to establish that they must be well-formed after the change.

We could also have another template considering the use of the *select* transformation, which would be similar to the compositional one, as *select* transformations do not really transform assets, but simply select them. For this reason, we only present a template dealing with *ifdefs*. We do not present the formalisation of the Remove feature transformational template, but it is similar to the Remove feature compositional template formalisation illustrated in Section 4.1. Nevertheless, the languages used to represent the CK and assets are different because we allow CKs with transformations and *ifdef* blocks. Consequently, we cannot use the compositional template here. Moreover, we prove the template shown in Fig. 6 by induction over the CK, since we deal with a recursive semantics function. In contrast, the compositional template proof requires no induction.

4.3. General templates

In this section, we present templates which are CK language independent. The previously introduced templates are not general because they specify concrete CK changes. So, we need to represent these changes with concrete languages.

In the following scenarios, the CK does not change; only the FM or the AM. So, we can abstract from the CK structure. Consequently, these templates are compatible with any CK notation, including both compositional and transformational CKs. We first introduce the Change asset template, which deals with changes only to implementation files. This template is a particular case of the AM partial refinement compositionality introduced in Section 3.2.2. Moreover, we also have templates to deal with changes only to the FM. These would be particular cases of the FM compositionality (see Section 3.2.1).

Change asset

Developers modify source files in many contexts, such as when fixing bugs or implementing new features. In such situations, one possibly does not desire to preserve behaviour. Thus, this is often a partially safe evolution scenario, since products that contain the changed asset might not preserve behaviour. 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 Fig. 7.

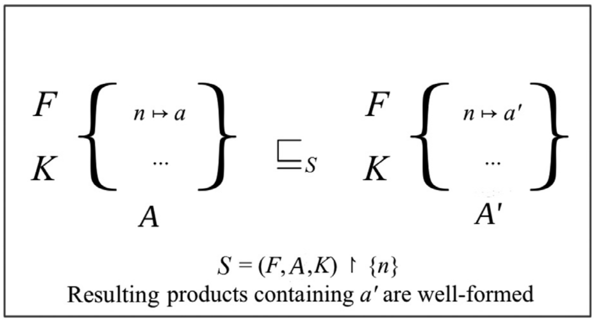

Fig. 7. Change asset partial refinement template.

To specify S for this case, we use another restriction operator. For an arbitrary product line (F, A, K), and set of asset names ns, we use $(F, A, K)\restriction ns$ to denote the subset of F configurations whose products do not contain assets from ns. Hence, in Fig. 7, S is defined as $(F, A, K)\restriction\{n\}$, which is the subset of configurations whose features are not implemented by the asset named n, which in this case is the a asset. Since products containing a' are possibly not refined, we cannot give any guarantees for them. There is also a well-formedness condition. Since we do not know which changes were performed to a, we need to demand well-formedness for products containing a'.

This 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). It could be the case that many assets change in a single evolution scenario, and we would still support developers because this is the same as applying the Change Asset template several times. Our transitivity property helps in such situations. 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 is introduced in what follows), for instance. Thus, developers could not only change assets, but also change their reference in the CK. As explained in Theorem 2, the guarantee is for the intersection of the products refined in both steps and this can be automatically calculated, as we define S for both templates.

The Change asset template also captures safe evolution scenarios. When one refines an asset, this template also matches. However, it would give less support than possible since we assume that the asset is being changed in a non behaviour-preserving way. The REFINE ASSET template (Neves et al., 2015) is more appropriate in this situation because it assumes that the asset changes and its behaviour is preserved, thus product line is safely evolved and gives guarantees for all products. In contrast, if the change impacts the product line behaviour, the REFINE ASSET template gives no support and developers should rather make use of the CHANGE ASSET template.

Transform Optional to Mandatory Feature

Feature types (*mandatory, optional, alternative* and *or*) may change during the evolution process. Some of these changes are refinements and others are not. For example, transforming a mandatory feature into an optional one is often a refinement, since the evolved product line would have more configurations than before, but we would still have the existing products, supporting existing users. This situation is addressed by previous work (Neves et al., 2015).

On the other hand, the opposite transformation, transforming an optional feature into mandatory, is often not a refinement. In this case, every product containing the parent of the transformed feature will also have the changed feature. Thus we would not be able to generate products without the changed feature but with its parent. For this reason, some users would not be supported, but others can be because products already containing the changed feature would be unaffected. This scenario is illustrated in Fig. 8. We give support for the original products that have O, because the only change applied to the initial product line was that O becomes mandatory. Furthermore, products without P are not affected, because they remain without O, as it is impossible to have O without having P

We have a condition in the template (Fig. 8) to guarantee that O can only be selected through the P selection, so there are no formula changing this condition. We state that we must be able to deduce the equation $O \Rightarrow P$ from F. So, it should not be possible to have O without P in the TRANSFORM OPTIONAL TO MANDATORY FEATURE template.

For this evolution scenario, we define S as the set of configurations that belong to the semantics of F, and satisfy the formula $O \vee \neg P$. This is expressed with the filter operator $\upharpoonleft$, which takes a FM F and a feature expression e and yields all configurations in F that satisfy e. This operator is the opposite of the restriction operator $\restriction$. We do not have any well-formedness condition for this template. This is not necessary because, in this particular case, we are able to prove that the resulting product line is well-formed. As there are no changes to assets in this case, we know that products remain well-formed. Moreover, there are no new products; it is just the case that some of the initial products do not exist anymore. Thus,

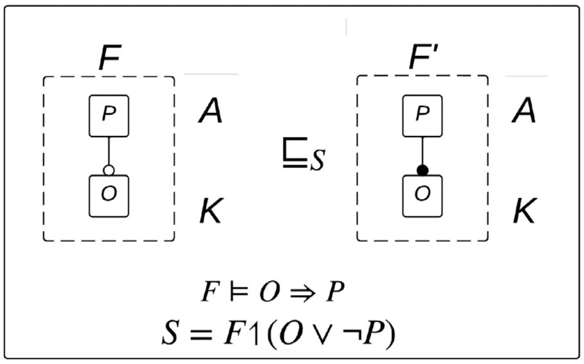

Fig. 8. TRANSFORM OPTIONAL TO MANDATORY FEATURE partial refinement template.

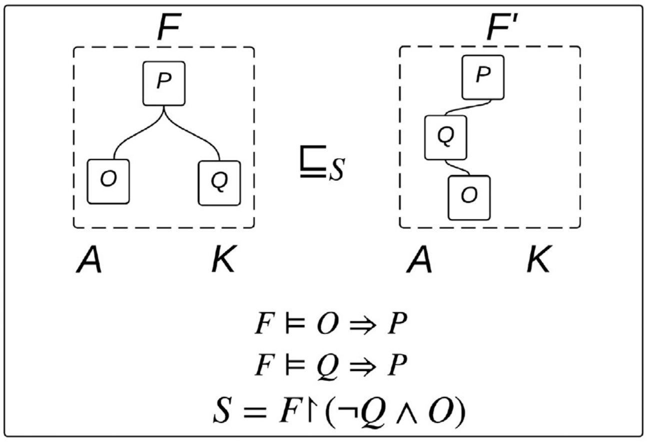

Fig. 9. MOVE FEATURE TO ITS SIBLING partial refinement template.

we had essentially to prove that all configurations belonging to F' semantics, also belong to F semantics.

Move Feature

Finally, we also consider changes to the FM regarding feature dependencies. During the evolution process, developers may want to move features in the FM. For example, a possible scenario is illustrated in Fig. 9. We have an initial FM F that has **at least** three features: P, Q and O. Feature P is the parent of Q and O. A change is performed and we then obtain F', where O is now Q's descendant. In this scenario, product configurations from the initial product line that do not have Q and have O are nonexistent in the resulting product line. In contrast, configurations that have O and Q do not suffer any impact, neither the ones that have P but do not have O. So, we define S as the set of configurations that belong to F semantics and do not satisfy the expression $\neg Q \wedge O$.

Besides that, the FM F, including cross-tree-constraints, should satisfy two expressions: $O \Rightarrow P$ and $Q \Rightarrow P$. This is necessary to guarantee that the FM formulae are not changing the relation between the features. So, we should be able to select O only if P is selected and this must hold for the entire FM. The same happens to Q and P. Consequently, both feature expressions should hold for the constraints of both FMs.

We do not specify any feature type in this template. This means that it applies no matter the types of features P, Q and O. However, depending on their types, we could have variations of this template that provide guarantees to different sets of products. For example, in two variations S is equal to all valid configurations, and, as a consequence, we would have refinement. This happens when only Q is mandatory, and when the three features are mandatory. In both cases, all products would have Q, so the expression $\neg Q \wedge O$, which should be satisfied by the configurations that are not refined, would not hold for any product. In any other scenario, S is not equal to F semantics. For instance, if all features are optional, products containing only P and O would not be generated in the resulting product line, as Q would need to be present, since it is the ascendant of O in F'.

We have just discussed a possible product line evolution scenario that consists of moving a feature in the FM, with the effect of changing feature dependencies. Nevertheless, as the FM structure is a tree, there are several potential scenarios of moving features in the scope of the tree. Although we have shown one possible move feature transformation, there are several other possibilities. The FM tree can be large, and features may be moved to a place far from its origin. These are just examples and any case that does not match these templates needs to be analysed separately. Moreover, for each situation the set of refined configurations S may vary.

For this reason, we do not have a single template to represent all possible move feature scenarios.

4.4. Discussion

We derived the templates by adapting a catalogue of safe evolution templates (Borba et al., 2012; Neves et al., 2015) for situations where not all products are refined by products in the evolved product line. For instance, the Change Asset template in Fig. 7 essentially adapts the Refine Asset Neves et al. (2015) 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. The set S of configurations refined depends on each scenario. For instance, if we change the behaviour of an asset present in every product, S can be the empty set. This would mean that we cannot provide any guarantee. This is not an ideal situation, but we are not aware of how to avoid it. Eventually, developers need to perform changes that affect all products. We revisit this topic on Section 6.

Verifying completeness of the templates and proposing a minimal set are part of our future work. We would also possibly need more templates, but we already cover several situations, like feature removals, CK line additions and removals, changes to the implementation, among others. If it is not possible to obtain absolute completeness, we could then establish a relative completeness by showing that the templates are expressive enough to transform an arbitrary product line to a reduced normal form.

Besides the introduced templates, we have others that are not presented here for brevity and can be found in our online appendix (Partial refinement theory website). For this reason, we show the full list of templates in Table 1. In addition to the Remove Feature and the general templates, we also have templates to deal with changes to the CK that are also formalised in both CK compositional and transformational notations. Finally, we have templates to cover asset removals and additions that were implemented in both transformational and compositional theories.

We do not present proofs of the general templates, but they are available in our online appendix (Partial refinement theory website). The templates that deal with changes to the FM only are relatively simple to prove, as they do not deal with any change to the implementation, so we basically need to prove that the set S of refined product configurations can be obtained from the initial and evolved FM semantics. We also guarantee that the evolved FM is well-formed for these templates. We do not need to deal with CK semantics and AM peculiarities. Regarding the Change asset template, it is a particular case of the AM partial refinement compositionality (see Section 3.2.2). Thus, we make use of the existing AM partial refinement notion to prove this template.

In general, our aim is to give support for several product line languages to increase the partial refinement concept expressiveness. When a particular product line element is changed, we detail such element using a particular language and try to understand which languages are possible to use. For the CK, we show that both compositional and transformational notations are compatible with the templates. Otherwise, we specify and prove the template in a more general level assuming that the elements are black boxes and they can thus be of any notation.

Table 1
Full template list.

Template	Compositional	Transformational	General
Remove Feature	✓	✓	-
Add CK Lines	✓	✓	-
Remove CK Lines	✓	✓	-
Change CK Line	✓	✓	-
Add Assets	✓	✓	-
Remove Assets	✓	✓	-
Change Asset	-	-	✓
Optional to Mandatory	-	-	✓
Move Feature	-	-	✓

5. PVS encoding

We present a partial refinement theory in Section 3 with definitions, properties and theorems about partial refinement of product lines, allowing reasoning over partially safe evolution of product lines. To guarantee soundness, we used a proof assistant to avoid human mistakes in manual proofs. This way, we can be confident that our proofs are correct, and the way that theories and proofs were mechanized also made the process less time consuming, as manual proofs could take much longer. We choose to use PVS, which provides a specification language, a type checker and an interactive theorem prover, mainly due to building the partial refinement theory on top of the refinement theory, which has been mechanized in PVS on previous works (Borba et al., 2012; Teixeira et al., 2015a).

To understand how the theory is encoded in PVS and relate to the product line refinement theory, we discuss the impact and the extension of the product line refinement theory in PVS (Partial refinement theory website). In Fig. 10, we show the dependencies among theories (PVS modules) and their hierarchy. Although we do not show the entire hierarchy here, all PVS files and proofs can be found online (Partial refinement theory website). The new theories created in this work due to the inclusion of partial refinement are highlighted in light grey colour. As the partial refinement concept builds on the existing product line theory and concepts (white PVS modules), all assumptions existing there, like asset set refinement preorder (Borba et al., 2012), are also required here. On top of that, we have extra specific assumptions for partial refinement, like the axioms presented in Section 3.2.2. We prove all new properties to guarantee that they are valid and consistent with the existing theory. In the remainder of this section, we explain each theory in more detail. Some of them deal with general notions of FM, AM and CK (*PartialRefBasics, PartialRefDefault, PartialRefWeaker, PartialRefinement* and *PartialAMCompositionality*), and ideally they would be valid with any product line definition that has the three elements. These five theories are parameterised with respect to FM, Asset, Asset Name and CK types, and also FM and CK semantics functions. Thus, one can instantiate them with concrete languages for the three product line elements and implement semantics functions. One needs to provide concrete notions, and semantics functions for the FM and CK. The other two theories are specific for compositional and transformational CK, respectively:

- **PartialRefBasics**: in this theory, partial refinement and weaker equivalence notions for the FM, AM and CK are defined. It imports the preexisting refinement theory that abstractly define these basic types representing the three main elements of the product line (Definition 5, Definition 7 and Definition 10).
- **PartialRefDefault**: this theory contains the main partial refinement definition (Definition 4). It uses **PartialRefBasics**, as we analyse whether the transformations applied to the FM and the CK in separate lead to product line partial refinement. Furthermore, we also reason about refinement and partial refinement transformations being applied consecutively. So, Theorems 11 and 12 are defined in this theory.
- **PartialRefWeaker**: this is analogous to **PartialRefDefault**, but here we deal with the weaker partial refinement notion (Definition 11).
- **PartialRefinement**: in this theory, we relate the previously presented definitions (default and weaker). Basically, here we establish that if the function f in the weaker definition is the

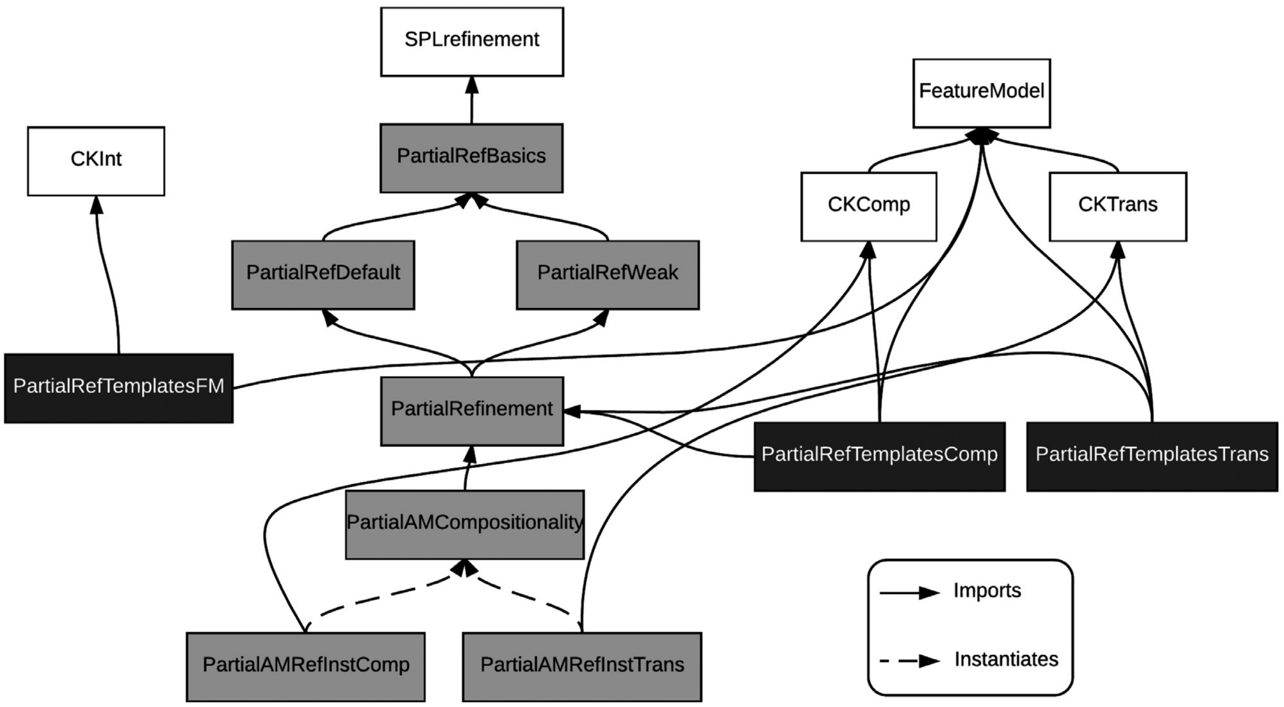

Fig. 10. PVS partial refinement theory.

identity function, this definition is equivalent to the relation in Definition 4.

- **PartialAMCompositionality**: as we discussed in Section 3.2.2, we assume the existence of an evaluation function to reason about AM compositionality. For this reason, this is expressed in a separate theory, as we do not need the evaluation function for the other concepts. We do not allow the evaluation function to be arbitrarily defined. As discussed in Section 3.2.2, it must obey a set of constraints, like not generating assets not present in the AM of the product line being evaluated.
- **PartialAMRefInstComp**: this is an instantiation of **PartialAM-Compositionality**, where we deal with the general CK notation. This theory is essential to certify that assumptions made regarding an arbitrary CK would be valid for the compositional CK. So, we prove that Axioms 2,1 and 3 also hold for compositional CKs.
- **PartialAMRefInstTrans**: analogous to **PartialAMRefInstComp**, but deals with transformational CKs.

In Fig. 10, we also show the theories that specify templates. We highlight in black colour (white font) the template theories. As we discussed, we have three template categories and they vary mainly according to the CK notation used. Consequently, we have three different template theories. For each template developers should obey the syntactic and conditions predicates like the ones presented for the Remove Feature template in Section 4.1. We also determine, for each case, the S set of refined product configurations. In the following, we explain each template theory for partial refinement:

- **PartialRefTemplatesComp**: this theory comprises the templates proposed in Section 4.1. It uses the concrete notions for FM (**FeatureModel** theory) and CK (**CKComp** theory). Since we are defining partial refinement templates, these theories all import the **PartialRefinement** theory.
- **PartialRefTemplatesTrans**: this is analogous to **PartialRefTemplatesComp**, but it deals with transformational CKs and uses the **CKtrans** theory instead of **CKComp**.
- **PartialRefTemplatesFM**: this theory corresponds to the templates presented in Section 4.3, except the Change Asset template that is specified together with the AM compositionality theory. This template is in a separate place because it assumes an *eval* function (see Section 3.2.2), which is part of the CK semantics. All the other templates deal with changes to the FM only, so all specific cases of the FM weaker equivalence compositionality are specified here. These templates assume a specific notation for the FM, that is structured as a tree. So, we use the concrete FM theory and the intermediate CK theory **CKint**, since the templates do not specify any CK language and would be compatible with any concrete CK that is a instantiation of **CKint**.

Templates Formalisation

All of the templates presented in this paper were encoded and proved in the PVS system. However, we do not present all specifications and proofs here, but all PVS files are available online (Partial refinement theory website). In this section, to illustrate our formalisation approach, we present the Remove feature template formalisation. The template we formalise here is actually more general than the template presented in Fig. 5, since it allows more than one line of the CK to be removed and also more than one asset from the AM. We illustrate as if the feature appears only in one CK line and is implemented by one implementation artefact just to make it more readable. To specify partially safe templates, we follow the same strategy found in previous works (Gheyi et al., 2005; Borba et al., 2012; Neves et al., 2015; Teixeira et al., 2015a). So, we first define the *syntax* and *conditions* predicates to encode the information present in the template. All syntactic similarities and differences regarding the initial and final product lines form the *syntax* predicate. In the Remove feature template, the three product line elements are presented in detail, so we define syntactic rules for all of them. Preconditions like well-formedness rules are specified with the *conditions* predicate.

To specify the *syntax* predicate, we make use of preexisting functions defined for concrete FM and CK PVS encoding. For example, the FM we are dealing with has a set of features and a set of formulae. So, for the Remove feature template, we state that F' (*fm*2 in the formalisation) formulae are all in F (*fm*1 in the formalisation), except those that have O. This is formalised as $formulae(fm2) = remove(O, formulae(fm1))$, and *remove*(*O*, *formulae*(*fm*1)) expands to $\{f\colon Formulae \mid f \in formulae(fm1) \wedge O \notin names(f)\}$. It would not make sense to allow these formulae to be part of F', since the O feature is removed. So, this guarantees that the O feature does not appear in any formula in F. Besides that, we also require that features from F' are exactly the ones from F, except for O. As Fig. 5 shows, P and O need to be features from the initial FM. As a consequence of the second condition, P is also in F'.

We also describe the AMs and CKs. The removed feature does not need to be implemented by one asset only, nor be present in only one expression in the CK. We assume that several items in the CK and in the AM may be removed, and we represent these two sets with the *its* and *pairs* variables, respectively. Thus, the specification is actually more general than the template shown in Fig. 5. Basically, the initial AM must be an extension of the final one, with the AM *pairs*. The override $\oplus$ operator can be simplified to $pairs \cup (am2 - dom(pairs))$, where $am2 - dom(pairs)$ is the set of pairs that belong to *am*2 whose names are not in *dom*(*pairs*). The CK is represented as a set of items in the compositional language, so we say that K has every item from K' and also the removed items in *its*. Finally, we also need to certify that every configuration satisfies the $O \Rightarrow P$ expression, as this express the parenthood expressed in the FM.

$$\begin{aligned}
&syntax(fm1, fm2, am1, am2, ck1, ck2, P, O, its, pairs):\\
&bool = formulae(fm2) = remove(O, formulae(fm1)) \wedge\\
&\qquad features(fm2) = remove(O, features(fm1)) \wedge\\
&\qquad P \in features(fm1) \wedge\\
&\qquad O \in features(fm1) \wedge\\
&\qquad am1 = am2 \oplus pairs \wedge\\
&\qquad ck1 = ck2 \cup its \wedge\\
&\qquad \forall c \in [[fm1]] \cdot sat(O \Rightarrow P, c)
\end{aligned}$$

We also define preconditions. The first one is to require S to be $F{\restriction}O$, which represents the set of configurations that are in F semantics, but do not satisfy O. It is also necessary to make sure that every expression from *its* implies O, which is represented by the e' variable in Fig. 5. Regarding the CK, it is also required that O does not appear in other CK lines. So, we establish that configurations from the initial FM satisfy expressions from *its* if and only if they have the feature O. The second condition is related to well-formedness. Developers must be sure that initial products containing O implementation remain well-formed. Finally, we have a condition regarding *its* and *pairs*. We require that the remaining features do not have their implementation removed, by guaranteeing that if an item does not belong to *its*, its respective assets, obtained by *assets*(*item*), are not in *pairs*, and consequently not removed.

$$\begin{aligned}
&conditions(fm1, S, its, pairs, P, O, ck, ck2, am2) : bool =\\
&\quad S = F \restriction O \wedge\\
&\quad \forall c \in [[fm1]]\cdot\\
&\qquad \forall exp \in exps(ck)\cdot\\
&\qquad\quad sat(exp, c) \Rightarrow exp \in exps(its) \Leftrightarrow sat(O, c) \wedge\\
&\qquad c \notin S \Rightarrow wf([[ck2]]_c^{am2}) \wedge\\
&\quad \forall item \in ck \cdot item \notin its \Rightarrow\\
&\qquad\quad \forall an \in assets(item) \cdot an \notin dom(pairs)
\end{aligned}$$

The template is already encoded, but we do not prove it directly. We define a strategy for an step-wise proof. First, we prove that configurations in S do not satisfy any expression in *its*. This guarantees that, when evaluating the CK, items associated with the O feature are not included, and consequently artefacts that implement O are not also present. This is specified in Lemma 1. The *evalCK* function yields CK items whose feature expressions are satisfied in the configuration c.

Lemma 1 (Items from its are not included). *For product lines $L = (F, A, K)$ and $L' = (F', A', K')$, a set of configurations S, a set of items its, an AM pairs and features P and O, if syntax(F, F', A, A', K, K', P, O, its, pairs) and conditions(F, its, pairs, P, O, K) hold, then*

$$\forall c \in S \cdot \forall item \in evalCK(K, c) \cdot item \notin its$$

We also introduce Lemma 2, to establish that, for products in S, assets resulting from the evaluation of the initial CK do not belong to the removed assets from *pairs* (note that *eval*(*K*, *c*) yields all asset names mapped to feature expressions that are satisfied according to the configuration c). This means that assets implementing the removed feature are not present in the evolved CK. This lemma is related to Lemma 1, where we show that the evolved CK does not have any expression involving the removed feature. Although we do not present in detail here, all definitions used in this proof can be found in our Git repository.[6]

Lemma 2 (Assets from. *pairs* ***are not included)*** *For product lines $L = (F, A, K)$ and $L' = (F', A', K')$, a set of configurations S, a set of items its, an AM pairs and features P and O, if syntax(F, F', A, A', K, K', P, O, its, pairs) and conditions(F, its, pairs, P, O, K) hold, then*

$$\forall c \in S \cdot \forall an \in eval(K, c) \cdot an \notin dom(pairs)$$

We are now able to prove that removing a feature, given the *syntax* and *conditions* predicates previously established, leads to product line partial refinement. This is formalised in Theorem 13. Essentially, when a feature is entirely removed from a product line, and no elements regarding the remaining features are removed, we say that the evolved product line partially refines the initial one for configurations that do not have the removed feature.

Theorem 13 (Removing a feature is a partial refinement)**.** *For product lines $L = (F, A, K)$ and $L' = (F', A', K')$, a set of configurations S, a set of items its, an AM pairs and features P and O, if syntax(F, F', A, A', K, K', P, O, its, pairs) and conditions(F, its, pairs, P, O, K) hold and $\forall c \notin S \cdot wf([[K']]_c^{A'})$, then $L \sqsubseteq_S L'$, where $L' = (F', A', K')$.*

Proof. We have to prove that $L \sqsubseteq_S L'$, which, according to Definition 4, expands to

$$S \subseteq [[F]] \wedge S \subseteq [[F']] \tag{27}$$

and

$$\forall c \in S \cdot [[K]]_c^A \sqsubseteq [[K']]_c^{A'} \tag{28}$$

To prove Predicate (27), we first expand the restriction operator $\restriction$ in our assumption about S, which leads us to $\forall c \in S \cdot c \in [[F]] \wedge \neg sat(O, c)$. So, we can conclude that $S \subseteq [[F]]$. To prove that $S \subseteq [[F']]$, we expand the FM semantics function $[[\_]]$, and we have to prove that all features and formulae in F' belong to F. $[[F']]$ expands to $\{c\colon Configuration \mid satImpConsts(F', c) \wedge satExpConsts(F', c)\}$. Expanding these two predicates, we have to prove that:

$$\forall c \in S \cdot \forall (n : Name) \in c \cdot n \in features(F') \tag{29}$$

and

$$\forall c \in S \cdot \forall (f : Formula) \in formulae(F') \cdot sat(f, c) \tag{30}$$

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

From S definition, we have that $S \subseteq [[F]]$, which amounts to $\forall c \in S \cdot \forall (n\colon Name) \in c \cdot n \in features(F)$. So, to prove Predicate (29), we need to prove that n cannot be O. Otherwise, since it belongs to *features*(F), it will also be in *features*(F'). Also from S definition, we conclude that c does not satisfy O. So, O cannot be in c's names. Predicate (30) also holds because it holds for *formulae*(F) and *formulae*(F')⊆*formulae*(F).

We then need to prove Predicate (28). Assuming an arbitrary $c \in S$ and expanding the CK semantics function, this simplifies to $A < eval(K, c) > \sqsubseteq A' < eval(K', c) >$. The difference between K and K' is the set of items (*its*) that belong to K but not to K'. Expanding *eval*(K, c) results in *assets*(*evalCK*(K, c)). By using Lemma 1, we conclude that all CK items resulting from *evalCK*(K, c) do not belong to *its*. As the other items, except *its*, are equal, K and K' evaluation results in the same items for configurations in S. So, we have that $assets(evalCK(K, c)) = assets(evalCK(K', c))$.

The difference between A and A' refers to *pairs*. In Lemma 2, we show that there are no assets from *pairs* resulting from the evaluation of K. So, assets obtained by the image in A and A' are the same and do not refer to *pairs*, so it makes no difference in obtaining the image in A and A', which is formally expressed as $A < eval(K, c) >= A' < eval(K, c) >$. Given that $[[K]]_c^A = [[K']]_c^{A'}$ for an arbitrary c in S, the proof follows by asset set reflexivity (Borba et al., 2012).

According to previous work (Borba et al., 2012), an FM is well-formed if its formulas only refer to its features. If features not present in the FM are referred, the FM is not well-formed. We are able to prove that the final FM is well-formed because, since the initial FM is well-formed, and the only transformation is removing a feature, F' is also well-formed as we remove the formulas that refer to the removed features. Regarding the final product line well-formedness, we need to prove that all products are well-formed, according to the product line definition. There are two scenarios to be considered: if a product is in the scope of S, we guarantee its well-formedness because it is equal to an initial product, as we have just proved in this theorem. For products that do not belong to S, we make use of the condition requiring that all products in L' which are not in S should be well-formed. So, we are able to prove that L' is well-formed. □

6. Evaluation

Although we expect our partially safe evolution templates could be useful in a number of situations, it is important to gather empirical evidence so we can better understand how often they could be applied in practice. To do that, we perform a quantitative retrospective study by analysing two product lines, Linux[7] and Soletta.[8] Both projects are active on GitHub, the variability model is written in Kconfig, Makefiles are used to map features to their implementation and C is the main programming language used for source code files. Linux is a large and highly variable system that has been used in previous works (Israeli and Feitelson, 2010; Adams et al., 2008; Dintzner et al., 2017). Soletta is smaller and more recent, so we also chose to analyse this system to understand whether characteristics such as project size and number of commits have any influence in our analysis.

We try to find scenarios that match our templates by analysing commits from the two projects. In this section, we detail the data extraction process in Section 6.1, show the results for each template in Section 6.2, and discuss threats to validity in Section 6.3.

The purpose of our 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 could partially safe evolution templates be applicable in product line projects?

In order to answer this question, we automatically analyse commits from the Linux and Soletta projects, where each evolution scenario is composed of a commit pair: the initial and evolved product lines are the ones in two consecutive commits. We measure the number of occurrences of the proposed templates, since they represent partially safe evolution situations.

6.1. Setup

We use the FEVER tool (Dintzner et al., 2017) to identify template occurrence.[9] This tool, developed by Dintzner et al., is able to analyse commits from projects that use the Linux notation. FEVER takes a set of commits as input and collects information from them. Then, the differences from each pair of consecutive commits are processed and the resulting information is stored in a Neo4j database.[10] The tool discovers which Linux elements, such as the variability model, were changed in evolution scenarios. Additionally, changed files are automatically classified into source and non-source. To find occurrences of our templates, we query the database populated by FEVER to filter evolution scenarios by expressing the conditions for each template, such as whether they affect the FM. For instance, in the Change Asset template (Section 4.3), we ensure that only the code is modified. Thus, there are no changes to the FM and CK.

We manually check all evolution scenarios returned by the queries (except those representing changes only to the implementation, as the number is extremely high) to make sure they really match the templates, and also to reduce false positives. The tool can also have bugs, so the manual analysis is also important to mitigate the tool imprecision. To reduce false positives in Change Asset instances, we run a complementary analysis. Altogether, we analysed 67,310 evolution scenarios of the Linux Kernel from the database we had access to, and this corresponds to all commits between Linux versions 3.11 and 3.16. The first commit was performed on September 2nd of 2013 and the last one was on August 3rd of 2014, so this comprises roughly one year of development. We try to match each evolution scenario with one of the templates, based on their conditions, as explained in the following.

Remove Feature: scenarios that modify all three elements of the product line, removing elements. These three modifications must be correlated, as illustrated in Section 4.1. Thus, the removed mappings need to be associated with the removed feature in the FM. Similarly, the removed assets in the CK need also to be excluded from the implementation. These rules are detailed in Listing 4, using Neo4j query language. In the database, the *MappingEdit* and *FeatureEdit* nodes represent changes to the CK and FM, respectively. An *ArtefactEdit* is any file change. 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 establish 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 feature name in the FM needs to be the same name as the edited feature in the mapping change (CK) (Line 3). As the three parts are affected, we also state that there should be CK removals (Line 5). It would make no sense to allow source code artefact additions in a feature removal scenario, so we filter these cases (Line 7). We also verify if changes in the implementation are

[7] Linux Kernel repository available at http://github.com/torvalds/linux.
[8] Soletta project website: http://solettaproject.org/.
[9] http://github.com/NZR/SPLR-FEVER-Tool.
[10] Neo4j website http://neo4j.com.

```
 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  mapping.target_type=‘‘COMPILATION_UNIT’’ AND
 7  NOT (c)−−>(:ArtefactEdit{type:‘‘source’’,change:‘‘ADDED’’}) AND
 8  file.name=~(‘‘.*’’+
 9              substr(mapping.target,0,length(mapping.target)−2) +
10              ‘‘.*’’)
11 return distinct c
```

Listing 4. Remove feature Neo4j query.

related to changes in the mapping (Line 10). All distinct commits obeying these rules are then returned. In the Remove Feature template, we have a condition regarding well-formedness that, in our analysis, we assume to be true in every scenario.

This query is subject to false positives. Although the exact mapping between features and artefacts in Makefiles can be complex, FEVER relates each mapping change to one feature only, which may lead to imprecision. Additionally, features can be delimited with **#ifdef** annotations. So, when removing a feature, one can remove only an **#ifdef** block, without removing an entire source code file. Although we cover examples that deal with transformational CKs, this increases imprecision, since we cannot filter if only an **#ifdef** has been removed. The FEVER tool is not able to detect such change. To deal with such false positives, we perform manual analysis to guarantee their absence.

False negatives may arise due to special cases. For instance, the removed file not necessarily has the same name of the mapping target removed in the CK. Thus, this evolution scenario would not be found with this query because the last condition may not hold. Additionally, we also do not find scenarios that are compositions of feature removals and other changes. For instance, one could remove a feature and add a new one in the same commit. Errors in the data set can also lead to false negatives. We do not show queries for the other templates, but they follow a similar approach and are available in our online appendix (Partial refinement theory website). We should remind that false negatives do not affect our results, as they would actually increase our templates occurrence.

Change Asset: we classified an evolution scenario as a change asset instance when only the implementation changed. We filter commits that have at least one source file changed. It was also necessary to establish that the commit had no added or removed source files. Therefore, we only capture cases where the change is in source code. If only a non-code file is changed, such as a *.txt* file, we do not consider it a change asset instance.

Add Assets and Remove Assets: we classify evolution scenarios as instances of these templates when only the CK and implementation change. In the former, both changes must be additions. The files added to the implementation should be new and of type *source*, according to FEVER. For the Remove Assets template, the query is analogous. So, we only allow removals in the CK and implementation. Source files should be entirely removed and CK lines must also be excluded. Moreover, we have a similar condition to the last one in Listing 4 to guarantee that the changes are related. Developers might remove Makefile mappings and source files independently. Thus, we check whether the source file names appear in the affected CK lines. We do not consider any case in which the FM changes. This would actually consist in a feature addition or removal.

We are not aware of false positives that may arise due to the Add Assets and Remove Assets queries. Regarding false negatives, we do not find instances in which an added file has not exactly the same name of the added CK line. However, changing such condition would probably increase the false positives number.

Change CK Line, Add CK Line 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. Another reason is that the number of instances is considerably small for these templates. For all of them, we required that the implementation and FM elements must remain unchanged. We also identified the other templates in a similar way, and the results are presented next.

We can have false positives in these instances because we are filtering any mapping change. So, a number of them might not be of our interest. We could filter mapping additions, removals and modifications separately, but the FEVER tool uses the Git diff algorithm, which has an imprecise classification. We prefer to filter all changes and manually classify according to their types. We are not aware of false negatives due to the query, but they can occur due to data set problems.

6.2. Results

In this section, we discuss the results from the analysis of the Linux and Soletta systems. Linux and Soletta are similar with respect to the notation used for its elements, but vary in size and maturity level. These differences reflect in our results. For each project, we inform how often our templates could be applied in evolution scenarios. We also discuss threats to the validity of our results.

Although we classify templates into compositional and transformational in Section 4, both systems use a transformational CK notion. For instance, some feature removals actually affect **#ifdef** annotations and our compositional template is not compatible with such structure. So, precisely, these would rather match the transformational and general templates.

6.2.1. Linux kernel

We present the numbers of each template in Table 2. The Change Asset template could be often applied, matching almost 90% of the evolution analysed scenarios. By identifying commits that actually change only white spaces and permissions, we

Table 2
Template occurrence - Linux Kernel.

Template	Query returned	Excluded	Imprecision Source	Remaining
Change Asset (and possibly Refine Asset)	55,345	780	Query	54,565 (89.4%)
Add Assets	181	13	Tool	168 (0.27%)
Remove Assets	17	1	Tool	16 (0.02%)
Change CK Line	18	0	Query	18 (0.02%)
Add CK Lines	9	0	Query	9 (0.01%)
Remove CK Lines	12	0	Query	12 (0.02%)
Remove Feature	93	25	Query	68 (0.11%)

Table 3
Template occurrence summary - Linux Kernel.

Commit type	Number
Non-merge	67310
Merge	5413
Analysed	61897
Match our templates	55,676 (89.94%)
Not match any template	6221 (10.06%)

exclude 780 scenarios. We try to reduce the query imprecision by analysing commit messages, which suggest that partial refinement occur more frequently than refinement scenarios. The other templates had considerably lower numbers. The Add Assets query returned 181 instances, but, only 168 were considered, since the other 13 do not match the template due to data set problems which will be explained later. Similarly, the Remove Assets template had 16 instances only. The templates that deal with changes to the CK (Change CK Line, Add CK Lines and Remove CK Lines) had 18, 9 and 12 instances. Finally, the Remove Feature query returned 93 instances and after manual analysis 68 remain valid. We believe that there are no problems due to data set in this template, but the query is not precise enough.

The numbers in Table 2 are lower bounds of the cases we could confirm. We provide a summary of our analysis in Table 3. From the 67,310 commits, 5413 are merge commits, which are discarded by the tool because they correspond to integration, not evolution, scenarios. Hence, we could give support for 89.94% of the cases altogether, as shown in Table 3.

There are, in fact, 6221 instances in the Linux system that do not match any of our current templates, which could include, for instance, commits that only change feature dependencies in the FM, commits that represent feature additions (which change the three elements of the product line), or even refinement scenarios such as feature renaming. As discussed in Section 4, the proposed templates were adapted from product line refinement templates proposed in previous works (Neves et al., 2015). So, we aim to investigate these 6221 instances in more detail, and, if necessary, propose new templates to deal with them as well.

We are not able to identify instances of all our current templates with the FEVER tool. For example, we found no instance of the Transform Optional to Mandatory template. The Linux Kconfig model does not provide a clear feature classification into *optional, mandatory, alternative* and *or*. So, the current version of the tool is not able to inform feature types. This would require a deeper interpretation of the Kconfig model, and possibly adding and improving modules of the FEVER tool.

In the following, for each template, we discuss in more detail the number of instances found. We also provide examples of evolution scenarios that match our templates and show examples that were excluded due to problems related to query and data set, among others.

Change Asset

According to Dintzner et al. (2017), 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, with 55,345 instances, which corresponds to almost 90% of the evolution scenarios analysed. This might be due to Linux maturity level, and also to the fine granularity of the commits observed in the analysed period.

The Change Asset template matches any implementation change. Since we do not have control over these changes (they could be refinements or non-refinements, for instance), we use auxiliary tools to have a more precise idea of changes. Knowing precisely what are the changes made in Change Asset instances is extremely important because they correspond to almost 90% of our scenarios found and this could affect the applicability of the Change Asset template. For instance, those scenarios classified as refinement should not be considered instances of the Change Asset template. As number of occurrences is extremely high, we could not manually verify all cases. Thus, a number of these occurrences might be full refinements. By manually analysing 50 Change Asset instances (randomly 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, for instance, by modifying *if-then-else* conditions. Based on this analysis, we raise the hypothesis that partial refinements occur more frequently, and this makes the Change Asset template far more frequently applicable than the Refine Asset template (Neves et al., 2015).

An example of a Change Asset scenario is the pair formed by commit *2627b7e15c*[11] and its predecessor. In Listing 5, a developer removes the call to the *ip_vs_conn_drop_conntrack* function (we used the - symbol to indicate line removal) to avoid a crash, as he explains in the message. This is the only change; the other lines remain untouched. So, we consider this example to be a partial refinement, as there is a clear intention to change the feature behaviour by solving a bug. Moreover, regardless of the commit message, function call removals are often not refinement transformations (Borba et al., 2004; Cornélio et al., 2010). Unless the function has a void behaviour, the resulting program tends not to have a compatible behaviour to the initial one. In this example, products not containing the *net/netfilter/ipvs/ip_vs_conn.c* file are refined according to the set of products *S* specified in the Change Asset template.

To better understand the type of changes in the 55,345 commits returned by the query, we analyse commit messages for identifying terms that suggest that changes are not behaviour preserving. Using *Lucene*,[12] a natural language processing tool we rank every term in the commit messages according to its frequency in the text formed by concatenation all messages. We then select a number of best ranked terms and count the number of messages in which they appear. So we are able to know the number of commit messages that had each term.

Lucene ranked 209.328 terms that were found in the 55,345 messages. We can see from Table 4 that *Fix* and *Bug* occupy the

[11] Change Asset example: http://github.com/torvalds/linux/commit/2627b7e15c. Jul 8, 2014; version v3.16-rc5.

[12] Lucene website: http://lucene.apache.org/.

```
if (cp->flags & IP_VS_CONN_F_NFCT) {
-    ip_vs_conn_drop_conntrack(cp);
/* Do not access conntracks during subsys cleanup because
nf_conntrack_find_get can not be used afterconntrack cleanup
for the net.*/
...
```

Listing 5. An excerpt of “net/netfilter/ipvs/ip_vs_conn.c”.

Table 4
Frequent terms in Change Asset commit messages.

Term	Frequency	Rank
Use	12,609 (23.11%)	1
Fix	11,836 (21.69%)	2
Patch	9921 (18.18%)	3
Add	9916 (18.17%)	4
Remove	8352 (15.31%)	8
Error	4200 (7.69%)	41
Change	4131 (7.57%)	42
Bug	1870 (3.43%)	146
Failure	1228 (2.25%)	267
Rename	1111 (2.03%)	305
Modify	431 (0.79%)	954
Refactor	422 (0.77%)	976

2nd and 146th positions, respectively. This suggests that a great number of the Change Asset instances are bug fix scenarios. *Patch* is the 3rd most used term, which also suggests the high presence of bug fixes or general improvements, which may not be code refinements. Other words that might suggest the presence of product line refinement changes do not seem as frequent, like *Rename* and *Refactor*. This only gives a general idea, but we cannot be sure about the exact changes performed in Change Asset without analysing the code. Messages may not be well-written, or might be incomplete. Developers often do not explain in detail their commits and surely express differently their ideas, so this is just an approximation. Being conservative and considering that only the documents containing *Bug* and *Fix* represent partial refinement scenarios and all the others are refinement, we then have 12,680 Change Asset instances instead of 55,345. The number decreases considerably, but we would still be able to support 22% of the analysed scenarios.

The Lucene tool automatically excludes terms that are not of our interest, like prepositions, pronouns, among others. We also configured the tool to ignore others terms, such as *signed* and *off*, which are present at the end of every commit message and just pollute the rank. Besides analysing the rank of every single term, Lucene also allows us to search for specific expressions, for example, the number of messages that contain *bug* and *fix*, and including other possible terminations, like *fixes*. This provides a more powerful search than the single term one, but there is no ranking in this case. We found similar results by looking for expressions involving terms, and the results are available in our website (Partial refinement theory website).

In addition to analysing Change Asset commit messages, we also identified commits that only change spacing in code files by using Conflicts Analyser,[13] an open source tool that classifies conflicts according to a set of patterns (Accioly, 2015). Although we are not dealing with conflicts, the tool identifies differences between source code files. So, for every Change Asset instance, we compare the initial and final files. From the 55,345 commits returned in the Change Asset query, 777 only change white spaces, so these can be considered product line refinements, which corresponds to approximately 1,4% of the instances for the Change Asset template. We consider this number to be high, but it depends on project development practices. In the Linux case, changes have fine granularity, so this seems to be common. Commit *2055fb41ea*[14] is one of the instances found. In this commit, a line break is added before an *if* statement. No other changes are performed.

Our template could still be applied in this situation, because we do not make any restrictions to the changed artefact. However, one should rather make use of the Refine Asset template, as it gives guarantees that all products are refined, differently from the Change Asset template, that assumes that the asset is not refined and gives behaviour preservation guarantees for only a subset of the existing products. For this reason, we exclude these instances. The other three excluded instances are permission changes. For example, commit *186026874c*[15] changes the permission code of a *C* source file from *755* to *644*. In the Git version control system, which we deal with, permission changes may be committed in projects whose configuration file has the *filemode* parameter set to *true*, like the Linux Kernel.

In summary, this analysis indicates that most evolution scenarios are partial refinements. Our manual analysis confirms this hypothesis, given that we found 43 non-refinements out of 50 evolution scenarios. As the data set is really huge (55345 Change Asset) scenarios, it is impossible to analyse all of them manually. Using other tools to have a better understanding of the code is part of our future work.

Adding, Removing and Changing CK Lines

In our sample, we found 18, 9 and 12 scenarios that respectively correspond to mapping changes, additions and removals. We manually checked and confirmed the 39 instances. The numbers regarding these templates are not high because modifications focusing only on the mapping rarely occur (Kröher and Schmid, 2017), so the Change CK Line, Add CK Lines and Remove CK Lines templates have a lower frequency when compared to others, such as Add Assets. This might happen because most commits modify at least one source code file and some of them also modify 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 or references to the FM. It is also possible that an evolution scenario captured by one of our templates corresponds to a longer sequence of commits. Since we try to match each commit pair separately with the templates, this would explain the low occurrence.

We provide an example of a Change CK Line instance in Listing 6, which shows the differences between

[13] Conflicts Analyser website: http://twiki.cin.ufpe.br/twiki/bin/view/SPG/ConflictPatterns.

[14] http://github.com/torvalds/linux/commit/2055fb41ea. Jun 20, 2014; version v3.16-rc3.

[15] http://github.com/torvalds/linux/commit/186026874c. Jul 2, 2014; version v3.16-rc4.

```
−obj−$(CONFIG_ARCH_DAVINCI_DA850) += davinci−cpufreq.o
+obj−$(CONFIG_ARCH_DAVINCI) += davinci−cpufreq.o
```

Listing 6. Changes made to “drivers/cpufreq/Makefile”.

```
+obj−$(CONFIG_SOC_EXYNOS5410) += clk−exynos5410.o
```

Listing 7. Changes made to “drivers/clk/samsung/Makefile”.

```
209 lines added
```

Listing 8. Changes made to “drivers/clk/samsung/clk-exynos5410.c ”.

```
33 lines added
```

Listing 9. Changes made to “include/dt-bindings/clock/exynos5410.h ”.

commit *5a90af67c2*[16] and its predecessor. The presence of the artefact *davinci-cpufreq.o* was conditioned to the activation of *CONFIG_ARCH_DAVINCI_DA850*. After the change, the *CONFIG_ARCH_DAVINCI* feature is mapped to this artefact instead. In the message, the author explains that this commit fixes a build error. In such situations, there are no changes to the FM and implementation; only the CK changes, as we stated in our query. In this case, our template guarantees that products without the *CONFIG_ARCH_DAVINCI_DA850* and *CONFIG_ARCH_DAVINCI* features are refined.

Each scenario is classified as compatible with one template only, except for the Add CK Lines, Change CK Line and Remove CK Lines templates. Since they were mined with the same query, we noticed that some scenarios actually had instances of more than one of 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 discussed.

Adding and Removing Assets

FEVER returned 181 instances of the Add Assets template, which were all manually checked to confirm that they really match the template. Due to the tool imprecision, 13 of them did not. Therefore, we exclude these instances and only 168 remain, which precisely match our conditions and are instances of the template. There are at least 16 assets removals. We did not investigate the reason for such a lower removal rate. The results might be different considering another interval and project.

In Listings 7, 8 and 9, we show a scenario that matches the Add Assets template.[17] Basically, a line is added to the Linux CK, to map the *CONFIG_SOC_EXYNOS5410* feature to the *exynos5410.o* asset. As this asset is new, the *clk-exynos5410.c* and *exynos5410.h* files are added to the implementation. So, as the Change CK Line template requires, there is no change to the FM in this case. Also, the changes in the CK and implementation need to be related, and we do not allow source file removals or modifications. The only change in this commit that is not listed here is regarding documentation, but we do not forbid any change to a non-source file.

As we have already discussed, we only classify as Add Assets instances, commits that only touch the implementation and CK. So, other artefacts like the variability model are not allowed to change. However, among the commits returned by FEVER for our Add Assets query, we found 1 commit that changes the Kbuild file, 11 that change the Kconfig, and another where additional CK lines change. In all these examples, there are additions to the Makefile mapping and implementation files. However, FEVER accidentally returned some instances that do not match our query. For example, commit *d3e6573c48*[18] and its predecessor change additional lines in the CK. However, by the tool imprecision, these extra changes do not appear in the data set, so it would not be possible to filter them.

Remove Feature

Our query returned 93 feature removal scenarios, but only 68 were classified as valid according to our templates. The other 25 non-removals are scenarios where the features were actually being moved. Commit messages help to identify these situations. All these excluded instances were found due to query imprecision, so we are not aware of imprecision in the FEVER tool for this template. One of the main problems is that we do not restrict source file modifications to deletions. Intuitively, one might argue that when a feature is removed from a product line, and no other changes are performed, we should have source file deletions, but not additions/modifications. This would be valid in a compositional product line development context, where one code artefact implements exclusively one feature. However, in the Linux system, developers can make use of *ifdefs*, so an artefact may implement more than one feature, and removing a feature from the code means basically removing the respective *ifdef*. For this reason, we allow source code removals and modifications, but we need to filter them manually. We already provide a valid example of the Remove Feature template instance in Section 2, and the 68 cases are available in our online appendix (Partial refinement theory website).

6.2.2. Soletta

Soletta is a development framework that makes writing software for IoT (Internet of Things) devices easier. By abstracting hardware and operating system details from a program, Soletta allows developers to write software for controlling actuators and sensors and communicating using standard technologies.

The same process used in Linux to find template instances was also applied to Soletta. By running FEVER and executing the queries, like the Remove Feature query detailed in Listing 4, we obtain the numbers regarding the Soletta project for the period

[16] CK line change commit: http://github.com/torvalds/linux/commit/5a90af67c2. July 10, 2014; version 3.16.

[17] Add assets commit: http://github.com/torvalds/linux/commit/e7ef0b632e. May 26, 2014; version 3.16-rc1.

[18] http://github.com/torvalds/linux/commit/d3e6573c48. Dec 24, 2013; version v3.15-rc1.

Table 5
Template occurrence - Soletta.

Template	Query returned	Excluded	Imprecision Source	Remaining
Change Asset (and possibly Refine Asset)	1496	0	Query	1496 (65%)
Add Assets	5	0	Tool	5 (0.22%)
Remove Assets	0	0	Tool	0 (0%)
Change CKLine	9	0	Query	9 (0.39%)
Add CK Lines	3	0	Query	3 (0.13%)
Remove CK Lines	0	0	Query	0 (0%)
Remove Feature	5	3	Query	2 (0.09%)

ranging from its creation in 26 Jun 2015 to 19 Apr 2016. This corresponds to almost one year of development. Altogether, we analysed 2300 commit pairs for the Soletta project.

Table 5 shows that the numbers are significantly lower when compared to Linux. This was expected, as this project is considerably smaller and we are analysing only 2300 commits. The Change Asset query returned 1496 instances, or 65% approximately. In the Linux project, this template corresponds to almost 90% of the commits. We believe that this difference is due to the commits granularity and project's maturity level. From the examples we observed, it should be the case that commits have a finer granularity in the Linux project than in Soletta. Thus, developers commit more often. It is expected, then, a higher number of Change Asset instances. Furthermore, Linux is considered a stable project, so changes are performed mostly to the code (Dintzner et al., 2017), and there are less feature additions, for example, than a more recent project like Soletta.

We only found five Add Assets instances. We expected to find more, as in the beginning the project might have a significant number of asset additions. Nevertheless, we suspect that most asset additions are also feature addition scenarios, where, apart from CK and implementation, the FM also changes. These instances would best match the Add New Optional Feature refinement template proposed in previous work Passos et al. (2015). There was no asset removal that matched our Remove Assets template. This is understandable because Soletta is relatively new.

The numbers for Change CK Line and Add CK Lines were proportionately high than in Linux. A possible explanation is that approximately 90% of the commits in Linux only change the implementation, contrasting with 65% in Soletta. Like Remove Assets, Remove CK Lines had 0 instances. We found five instances of the Remove Feature, but three were excluded. So, only two remain. This is also justifiable by the fact that this is a recent project, so we expected to find a greater number of additions instead of removals.

The results are summarised in Table 6. Surprisingly, we found only one merge commit for this period in Soletta, which could be justified by the rebase practice. Thus, 2299 commits were analysed. As we explained, the FEVER tool ignores merge commits. At least 65.89% of the evolution scenarios would match our templates. This rate is much lower than Linux, that is almost 90%. We believe that this difference is due to the projects maturity level. The Linux project is older and the analysed interval in Soletta includes the start of the project, that tends to have more feature additions, which would not match any of our templates. Another possibility is the granularity level for the changes. Linux commits have a finer granularity. So, each commit in Soletta possibly would be the result of applying in sequence more than one partially safe evolution template. Since we try to match each commit pair to a single template, we do not include such instance.

We have the same problems in Soletta and Linux regarding the results, as we use the same queries and FEVER in both projects. Change Asset instances are the most risky, since we do not precisely analyse the performed changes. So, we do not know the number of refinements, although there might be more partial refinements. Although we did not find any bug in the data set for the Add Assets and Remove Assets, we encountered such errors in Linux, so we do not eliminate this possibility. For the other four templates, the queries are not precise enough and we do not consider problems in the data set because we did not find any of them.

We excluded three Remove Feature instances from Soletta results. Commit *5293f12e59*[19] was excluded because it is basically a renaming, where the *FLOW_NODE_TYPE_FREEGEOIP* feature is renamed to *FLOW_NODE_TYPE_LOCATION*. So, this example is not considered to be a feature removal. We had similar situations in commits *8d2e8aeb2c* and *446bc7e43c*. We can see removals, but the features are actually renamed into others.

By running the Conflicts Analyser tool over Soletta Change Asset instances, we did not find any commit changing only white spaces. This may be due to two reasons: the number of processed commits (2300) and the project development practices. It might be the case that in the Linux project, commits have finer granularity and developers accumulate less changes before committing. We also perform the term analysis for commit messages in Soletta. Lucene found a total of 6028 terms in the 1496 messages. As shown in Table 7, the word *fix* occupies the top again, as the most used term, appearing in 356 documents. Other words found in the Linux analysis like *Add, Error, Remove, Bug* and *Refactoring* also appear, but in lower positions. Although this project is different, we can see some similarity to Linux rank. This result indicates that

Table 6
Template occurrence summary - Soletta.

Commit type	Number
Any	2300
Merge	1
Analysed	2299
Match our templates	1515 (65.89%)
Not match any template	785 (34.14%)

Table 7
Frequent terms in Change Asset commit messages.

Term	Frequency	Rank
Fix	356	1
Add	307	2
Error	95	22
Remove	78	40
Change	74	42
Bug	14	390
Failure	5	825
Modify	5	826
Refactoring	3	1152

[19] http://github.com/solettaproject/soletta/commit/5293f12e59. Sep 10, 2015; version v1_beta4.

bug fixes occur in a significant frequency in both projects, which is possibly higher than refactoring scenarios.

6.2.3. Conclusion

We analyse two product line systems to discover how often could our partially safe evolution templates be applicable. In summary, we found that the Change Asset template would be applicable to most evolution scenarios in both systems analysed (90% and 65%, respectively). The other templates, in contrast, had much lower occurrence rates (they only sum 0.45% and 0.83% altogether). Nevertheless, they could still be useful in some scenarios. We do not present results for the Transform Optional to Mandatory and Move Feature templates, but, based on our study, we expect them to have low occurrence rates as well. Our results confirm previous study (Dintzner et al., 2017) evidence that the FM and CK are not changed as often as the implementation. We should also remind that since the two systems analysed deal with transformational CKs, our compositional templates could not be applicable. However, the transformational ones are compatible and also the general ones, since they do not assume any particular CK language. It is part of our future work analysing product lines dealing with compositional CKs.

This evaluation extends our previous one (Sampaio et al., 2016) by considering annotation templates in addition to the compositional ones. We also analyse the Soletta project, which was not analysed before. We should highlight that while doing a deeper analysis, we discovered bugs in our previous Linux results. In particular, the number of commits we report here is slightly lower because, due to an error in our data set, we computed some examples twice. This has also directly impacted the results. Nevertheless, in general, our results have actually been improved. Most of them changed less than 0.1%, but the Change Asset had an increase in the number of instances of approximately 7%.

6.3. Threats to validity

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

Construct: As already mentioned in Section 6.2, to find occurrences of the Change Asset template, we search for any change in the implementation and do not analyse which type of modification was performed in the source file, thus possibly also retrieving commits which actually represent occurrences of the Refine Asset template Neves et al. (2015). Although we manually examined 50 commits and performed analysis using both Lucene and Conflicts Analyser, we cannot generalise to all commits. Regarding the term frequency analysis, it is superficial to make conclusions, specially considering that developers express differently their ideas. Moreover, we do not consider synonyms in that analysis, which could also lead to more precise results. Scenarios matching the other templates can be safe only in pathological cases, so we do not take them into consideration.

Internal: We should consider that the tool we use may have bugs. However, we perform manual analysis to eliminate all false positives, except for the Change Asset instances. For these, we perform a complementary analysis based on commit messages. Thus, we may only have false negatives, which would actually improve our results. For example, if in a Remove Feature scenario, FEVER does not capture that the three elements of the product line have been changed, our query will not return such scenario. We discussed such examples in the previous section.

False negatives rate also depends on the number of commits matched to an evolution scenario. We analysed 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.

Finally, we assume certain template conditions to be true, such as well-formedness. To reduce such imprecision we should use a strategy to verify well-formedness (Braz et al., 2016) to make sure that the template would be applicable. In our analysis, we make the open world assumption. Consequently, we analyse the scenarios locally instead of globally. For instance, in the Change Asset instances, we analyse the changed files but nothing else. It could be the case that the change seems to be a partial refinement (such as function call removals), but the global effect might be different. Thus, this is also a threat. In systems such as Linux it is not trivial to analyse changes globally and this is also part of our future work.

External: We only examined Soletta and a small part of the Linux repository history. Hence, we cannot generalise 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 analyse 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.

7. Related work

As discussed, this work extends the partial refinement theory for product lines (Sampaio et al., 2016), by presenting new properties (such as compositionality), and a broader evaluation study. We also rely on some other previous works. Alves et al. (2006) and Borba et al. (2012) define safe evolution for product lines. A product line is safely evolved when behaviour preservation holds for all initial products, and this is formalised through a refinement theory. Teixeira et al. (2015b) extended this work for product populations and multi product lines. With the aim of guiding developers in possible refinement scenarios, Neves et al. (2015) and Benbassat et al. (2016), among others, propose template catalogues to abstract safe evolution scenarios. Additionally, a product line of theories for reasoning about safe evolution of product lines was proposed by Teixeira et al. (2015a) to investigate and explore similarities between different languages that specify product line elements.

Dintzner et al. (2014) present a classification of feature changes as well as a tool named *FMDiff* to automatically analyse 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 analysing commits from the Linux repository history. Dintzner et al. (2017) also developed the FEVER tool, which we use in our evaluation, to enables the analysis of Linux commits and precisely informs which artefacts are affected by a change.

Thüm et al. (2009) classify FM edits into refactorings, specialisations, generalisations and arbitrary edits by using satisfiability solvers. Our work differs because it is not our goal to build a tool and to analyse the feature model structure only. However, our theory could be mechanised 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. (2015) propose a pattern catalogue 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 analysing 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 for the Linux context, our aim is to propose a new refinement theory and templates for product lines in general. They also suggest the need for a new theory to address non-refinement scenarios.

Nieke et al. (2016) analyse 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 analysing 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 analyse the variability model and do not propose a partial refinement theory, differently from our work.

Also in the context of the Linux system, Ziegler et al. (2016) present an approach to identify relationships between configuration options, which allows one to discover source files that might be affected due to a change in a configuration option. They found that most configuration options affect few files only, and a few options affect a significant number of files. This work is related to ours, as we also analyse changes in the Linux system. However, this is not the core of our work. We could use their approach to present more detail of the evolution scenarios to give an idea of the number of products affected by a change to a configuration option. Furthermore, Lotufo et al. (2010) provide a quantitative and qualitative analysis of the Linux product line. They discovered changes related to the FM, such as the number of features and the tree height, and how these changes influence in the Kconfig model complexity. While they focus on the Linux FM, we are interested in changes to the three elements of a product line.

Seidl et al. (2012) provide a remapping approach to keep product line artefacts after evolution. The authors classify changes to each product line element and inform developers possible inconsistencies that may arise. Our solution could be integrated to theirs in establishing other possible categories and supporting the inconsistency analysis, as we provide an impact analysis for a set of evolution scenarios.

Finally, there are several works (Ren et al., 2004; Zhang et al., 2012; Mongiovi, 2011) that propose change impact analysers for specific contexts. The approaches involve running tests to check whether behaviour is preserved after a change. Our work is also related to change impact analysis but we do not deal with any programming language in particular, so our discussion is more abstract. Moreover, our theory provides formal guarantees. Furthermore, we reason about changes not only to code, but also FMs and CKs, as we are dealing with product lines.

8. Conclusion and future work

In this work, we extend our partially safe evolution notion for product lines, by supporting developers when evolving the AM and CK providing compositionality properties for these two artefacts. We also analyse the compatibility of our templates with existing CK notions. As a result, we present partial refinement transformational templates to deal with CKs containing transformations, such as *preprocess*. Additionally, we extend our quantitative evaluation by analysing another product line, Soletta, and providing further information regarding Linux results. The motivation of this work is that product line evolution is a challenging and complex task. It is important to give guarantees during this process, even for a product subset only. Especially in highly configurable systems like the Linux Kernel, there are thousands of possible valid configurations and predicting whether products have their behaviour preserved is often hard.

As future work, we intend to expand our theory to deal with function transformations to specify refinement (see Section 3.4), and also prove that refinement and partial refinement commute. Additionally, we intend to correlate our work with previous work (Teixeira et al., 2015a) that defines a product line of product line theories. This way, we could apply our theory to itself and integrate our new theory to the existing product line of theories. Additionally, we could also improve our evaluation and further investigate Change Asset instances. Although we provide a commit messages analysis, it would be useful to know precisely the type of changes in each scenario, and also classify them in refinements and partial refinements.

Furthermore, we could provide the set of refined products in each scenario. This way, developers know, for instance, when considering the motivating example shown in Section 2, the exact set of products that had the *LEDS_RENESAS_TPU*. We did not evaluate the quality of our templates by means of the value of S in each scenario. As already discussed, if all products have this feature, we would have no products refined and this means an empty S. Consequently, developers would have no support. In contrast, if the feature is not present in a high number of products, the support tends to be much higher. So, this is a limitation and part of our future work.

Finally, we would like to develop a tool to support developers on software product line evolution. We could implement transformations described in the proposed templates to allow developers to automatically evolve product lines and inform the set of products refined. So, the tool would provide another layer of abstraction and use the partial refinement theory concepts in background.

Acknowledgments

This research was partially funded by INES 2.0,[20] FACEPE grants PRONEX APQ-0388-1.03/14 and APQ-0399-1.03/17, and CNPq grant 465614/2014-0. We also acknowledge support from CNPq (grants 409335/2016-9, 309172/2017-9, and 408516/2018-6), FACEPE (APQ-0570-1.03/14) and CAPES (129599/2016-01). Finally, we thank Nicolas Dintzner for the FEVER tool and support, members of the SPG for feedback.

References

Partial Refinement Theory Website. https://gabisampaio.github.io.
Accioly, P., 2015. Understanding conflicts arising from collaborative development. Proceedings of the International Conference on Software Engineering, 775–777.
Adams, B., De Schutter, K., Tromp, H., De Meuter, W., 2008. The evolution of the linux build system. Electron. Commun. EASST 8.
Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., Lucena, C., 2006. Refactoring product lines. Proceedings of the International Conference on Generative Programming: Concepts & Experiences, 201–210.
Apel, S., Batory, D., Kästner, C., Saake, G., 2013. Feature-oriented Software Product Lines: Concepts and Implementation. Springer.
Benbassat, F., Borba, P., Teixeira, L., 2016. Safe evolution of software product lines: feature extraction scenarios. Proceedings of the Brazilian Symposium on Software Components, Architectures and Reuse, 11–20.
Borba, P., Sampaio, A., Cavalcanti, A., Cornélio, M., 2004. Algebraic reasoning for object-oriented programming. Sci. Comput. Program 52 (1), 53–100.
Borba, P., Teixeira, L., Gheyi, R., 2012. A theory of software product line refinement. Theor. Comput. Sci. 455, 2–30.
Braz, L., Gheyi, R., Mongiovi, M., Ribeiro, M., Medeiros, F., Teixeira, L., 2016. A change-centric approach to compile configurable systems with #ifdefs. Proceedings of the International Conference on Generative Programming: Concepts & Experiences, 109–119.
Clements, P., Northrop, L., 2001. Software Product Lines: Practices and Patterns. Addison-Wesley,.
Cornélio, M., Cavalcanti, A., Sampaio, A., 2010. Sound refactorings. Sci. Comput. Program 75 (3), 106–133.

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

Dintzner, N., van Deursen, A., Pinzger, M., 2017. FEVER: An approach to analyze feature-oriented changes and artefact co-evolution in highly configurable systems. Emp. Softw. Eng. 1–48.

Dintzner, N., Van Deursen, A., Pinzger, M., 2014. Extracting feature model changes from the linux kernel using FMDiff. Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems, 22:1–22:8.

Gheyi, R., Massoni, T., Borba, P., 2005. A rigorous approach for proving model refactorings. Proceedings of the International Conference on Automated Software Engineering, 372–375.

Israeli, A., Feitelson, D.G., 2010. The linux kernel as a case study in software evolution. J. Syst. Softw. 83 (3), 485–501.

Kröher, C., Schmid, K., 2017. Towards a Better Understanding of Software Product Line Evolution. Softwaretechnik-Trends 37 (2).

Lotufo, R., She, S., Berger, T., Czarnecki, K., Wąsowski, A., 2010. Evolution of the linux kernel variability model. Proceedings of the International Systems and Software Product Line Conference, 136–150.

Mongiovi, M., 2011. Safira: a tool for evaluating behavior preservation. Proceedings of the International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, 213–214.

Neves, L., Borba, P., Alves, V., Turnes, L., Teixeira, L., Sena, D., Kulesza, U., 2015. Safe evolution templates for software product lines. J. Syst. Softw. 106, 42–58.

Nieke, M., Seidl, C., Schuster, S., 2016. Guaranteeing configuration validity in evolving software product lines. Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems, 73–80.

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

Passos, L., Teixeira, L., Dintzner, N., Apel, S., Wąsowski, A., Czarnecki, K., Borba, P., Guo, J., 2015. Coevolution of variability models and related software artifacts. Emp. Softw. Eng. 21 (4), 1744–1793.

Pohl, K., Böckle, G., van Der Linden, F.J., 2005. Software Product Line Engineering: Foundations, Principles and Techniques. Springer.

Ren, X., Shah, F., Tip, F., Ryder, B.G., Chesley, O., 2004. Chianti: a tool for change impact analysis of java programs. ACM SIGPLAN Notices 39 (10), 432–448.

Sampaio, G., Borba, P., Teixeira, L., 2016. Partially safe evolution of software product lines. Proceedings of the International Systems and Software Product Line Conference, 124–133.

Seidl, C., Heidenreich, F., Aßmann, U., 2012. Co-evolution of models and feature mapping in software product lines. Proceedings of the International Systems and Software Product Line Conference, 76–85.

Teixeira, L., Alves, V., Borba, P., Gheyi, R., 2015a. A product line of theories for reasoning about safe evolution of product lines. Proceedings of the International Systems and Software Product Line Conference.

Teixeira, L., Borba, P., Gheyi, R., 2015b. Safe evolution of product populations and multi product lines. Proceedings of the International Systems and Software Product Line Conference, 171–175.

Thüm, T., Batory, D., Kästner, C., 2009. Reasoning about edits to feature models. Proceedings of the International Conference on Software Engineering, 254–264.

Zhang, L., Kim, M., Khurshid, S., 2012. FaultTracer: a change impact and regression fault analysis tool for evolving java programs. Proceedings of the International Symposium on the Foundations of Software Engineering, 40.

Ziegler, A., Rothberg, V., Lohmann, D., 2016. Analyzing the impact of feature changes in linux. Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems, 25–32.

Gabriela Sampaio is a Ph.D. student at Imperial College London and a member of the Program Specification and Verification group. Her research interests are program verification, web development, software product lines and software reuse.

Paulo Borba is Professor of Software Development at the Informatics Center of the Federal University of Pernambuco, Brazil, where he leads the Software Productivity Group. His main research interests are in the following topics and their integration: software modularity, software product lines, and refactoring.

Leopoldo Teixeira received the doctoral degree in computer science from the Federal University of Pernambuco. He is a professor in the Informatics Center, Federal University of Pernambuco. His research interests include software product lines, refactorings, and formal methods.