






Safe Evolution of Product Lines Using Configuration Knowledge Laws

Leopoldo Teixeira¹ , Rohit Gheyi² , and Paulo Borba¹ 

¹ Federal University of Pernambuco, Recife, Brazil
{lmt, phmb}@cin.ufpe.br

² Federal University of Campina Grande, Campina Grande, Brazil
rohit@dsc.ufcg.edu.br

Abstract. When evolving a software product line, it is often important to ensure that we do it in a safe way, ensuring that the resulting product line remains well-formed and that the behavior of existing products is not affected. To ensure this, one usually has to analyze the different artifacts that constitute a product line, like feature models, configuration knowledge and assets. Manually analyzing these artifacts can be time-consuming and error prone, since a product line might consist of thousands of products. Existing works show that a non-negligible number of changes performed in commits deal only with the configuration knowledge, that is, the mapping between features and assets. This way, in this paper, we propose a set of algebraic laws, which correspond to bi-directional transformations for configuration knowledge models, that we can use to justify safe evolution of product lines, when only the configuration knowledge model changes. Using a theorem prover, we proved all laws sound with respect to a formal semantics. We also present a case study, where we use these laws to justify safe evolution scenarios of a non trivial industrial software product line.

Keywords: Safe evolution · Software product lines · Theorem proving

1 Introduction

A Software Product Line (SPL) is defined as a set of software systems built from a common asset base, that share common characteristics, but are sufficiently distinct from each other in terms of features [2, 23]. SPLs can bring significant productivity and time to market improvements [23]. Besides code and other kinds of assets, an SPL consists of different artifacts, such as Feature Models (FMs) [18] and Configuration Knowledge (CK) [11]. We use FMs to characterize products, defining what is common and variable through features, which are reusable requirements and characteristics [11]. We establish the relationship between features and concrete assets through the CK. Given a valid feature selection, CK evaluation yields the assets that build a product. When evolving an SPL, for example, to improve its design, it is important to make sure that

it remains well-formed, in the sense that it generates valid products, and that behavior of existing products is not affected.

Manually changing different parts to evolve an SPL requires effort, especially for checking necessary conditions to make sure the change is performed without impact to existing products, such as introducing bugs or changing behavior. Moreover, this process is time-consuming and can also introduce defects, compromising the promised benefits on other dimensions of cost and risk. Recent works investigating the evolution of highly configurable systems through their commit history show that the mapping between features and assets evolves independently [17, 21] of changes to the variability model or code. For instance, Gomes et al. [17] found that 122 out of 500 commits from the highly configurable system Soletta, only changed the CK, besides other commits that involve changes to multiple elements at the same time. In our earlier work, we formalized general theories for SPL refinement [8, 28, 33], that take into account FM, CK, and assets (Sect. 3), capturing the informal notions of safe and partially safe SPL evolution. Safe evolution states that the resulting SPL must be able to generate products that behaviorally match all of the original SPL products, while partially safe relates to preserving behavior of only a subset of the original products. Safe evolution, which is our focus in this work, even allows adding new products, as long as we maintain the behavior of the original products. Although formalized, these notions can also be costly to check, since it can be time consuming, besides error-prone, to reason over the semantics of all SPL products.

This way, in this paper, we propose a set of bi-directional, semantics-preserving transformations for compositional CK models, that associate enabling conditions to asset names (Sect. 2). We use such transformations to ease the reasoning over SPL maintenance when only the CK changes, justifying safe evolution. Therefore, there is no need for developers to do semantic reasoning, making their task more productive. We use the Prototype Verification System (PVS) [27] to specify and prove soundness of the transformations. We also use them to justify safe evolution scenarios of a non trivial industrial product line for automated test generation tools with approximately 32 KLOC. The formalization we present here not only avoid errors when manipulating these models, but also contributes towards the general SPL refinement theory [8], providing transformations that work with an instance of the concrete CK language previously defined.

The main contributions of this work are the following:

- We present a new CK equivalence notion and its associated compositionality results for the SPL refinement theory previously proposed [8] (Sect. 3);
- A set of transformations for compositional CK models associating feature expressions to asset names (Sect. 4);
- Soundness proofs for all laws using the formalized semantics (Sect. 5);
- A case study illustrating the applicability of the laws using evolution scenarios from a real SPL (Sect. 6).

The remainder of this work is organized as follows: In Sect. 2.1, we informally discuss the CK model used in this work, and in Sect. 2.2 we present its

formalization. In Sect. 7 we discuss related work, and we present conclusions and future work in Sect. 8.

2 Background

In this section, we explain the compositional CK model used in this work. Since its main purpose is to map features to assets, we first discuss FMs. FMs describe commonalities and variabilities in an SPL [18]. For instance, the FM in Fig. 1 describes a simplified mobile game SPL.¹ The root feature is **Rain of Fire**. Every product has an image loading policy, that depends on the memory size, so **Image Load Policy** is a mandatory feature. The loading policy is then either **On Demand** or during **Startup**, but not both. This denotes the mutually-exclusive relationship of *alternative* features. The game might also show clouds or not, so **Clouds** is an optional feature. Moreover, we can state cross-tree constraints over an FM using propositional logic. In this example, the formula below the FM states that whenever we select **Startup**, we must select **Clouds** as well. The FM describes product configurations, that is, valid feature selections that define different SPL products. So, the semantics of an FM denotes the set of SPL products. Formal representations of the semantics of an FM have been previously proposed [4, 29].

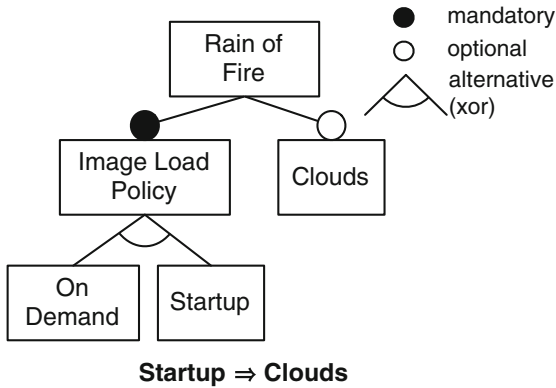


Fig. 1. FM of a simplified mobile game SPL.

2.1 Overview

The CK establishes the mapping between features and assets [11]. This knowledge can be mixed with implementation [3, 5], as in feature-oriented programming and preprocessing directives, or explicitly separated into a dedicated model [6]. In this work, we consider the latter: an artifact consisting on a list of configuration items, relating feature expressions to assets. These expressions, also

¹ Rain of fire mobile game, developed by *Meantime mobile creations*.

known as presence conditions, are propositional formulae, and enable expressive mappings of features into assets, instead of 1:1 mapping. The dedicated model also allows to be independent of the variability implementation language, so assets correspond to any kind of artifact.

Figure 2 illustrates the CK for the simplified mobile game SPL described by the FM in Fig. 1. Each row represents a configuration item, associating feature expressions with assets used to build products. In this example, assets are classes and aspects. For example, the second row establishes that if we select **Startup** or **On demand**, then **CommonImgLoad.java** should be part of the product. This class contains behavior shared by both image loading policies. So, we use an *or* feature expression to avoid repeating asset names in different rows.

Feature Expression	Assets
Rain of Fire	Game.java, GameScreen.java
On demand \vee Startup	CommonImgLoad.java
On demand	OnDemand.aj
Startup	StartUp.aj
Clouds	Clouds.java

Fig. 2. CK for the simplified mobile game SPL.

CK evaluation against a configuration yields the assets needed to build the corresponding product. For example, evaluating the CK of Fig. 2 against $\{\text{Rain of Fire, Image Load Policy, Startup, Clouds}\}$ yields the following set of assets: $\{\text{Game.java, StartUp.aj, GameScreen.java, CommonImgLoad.java, Clouds.java}\}$. This gives the basic intuition of the semantics of a CK [7,8].

As an SPL evolves, whether with new features and assets or with changes to existing assets, the internal structure of a CK might present problems. For example, in the CK of Fig. 2, we could replace the expression **Startup \vee On demand** with **Image Load Policy**. This way, if we add a new image loading policy, we do not need to change the expression in the CK. Duplication could also happen. For example, if we had used two rows, one for **Startup** and other for **On demand**, both associated to the same asset name **CommonImgLoad.java**. These issues, akin to bad smells [15], among others, can difficult the understanding of the model and its evolution.

2.2 Formalization

In this section, we present the formalization of CK models we have previously proposed [8] for models such as the one we present in the previous section. We use PVS [27] to specify this theory. Hereafter, we use well-known mathematical symbols instead of PVS keywords, such as SET, AND, EXISTS, and FORALL, for improving readability.

We define the configuration knowledge (CK) as a finite set of items. Items contain a feature expression and a set of assets. We specify them with *record type* declarations, enclosed in angle brackets. The following fragment specifies these and other types such as `FeatureName` and `AssetName`, which are *uninterpreted types*. This means that they only assume that they are disjoint from all other types. We define the `Configuration` type to represent product configurations as a set of feature names, representing the selected features.

```

FeatureName: TYPE
AssetName: TYPE
Configuration: TYPE =  $\mathcal{P}$ [FeatureName]
Item: TYPE= < exp:Formula, assets: $\mathcal{F}$ [AssetName] >
CK: TYPE=  $\mathcal{F}$ [Item]

```

Each CK item contains a feature expression, which is a propositional formula. Possible formulae are: feature name, negation, conjunction and implication. Other kinds can be derived from these. We represent these with PVS abstract datatypes [27], which we omit here for brevity.

To enable unambiguous references to assets, instead of considering that a SPL contains a set of assets, we assume a mapping from asset names to actual assets. Such an Asset Mapping (AM) corresponds to an environment of asset declarations. This allows conflicting assets in an SPL, for instance, two versions of the same class, that implement mutually exclusive features. The semantics of a given configuration knowledge K is a function that maps AMs and product configurations into finite sets (represented by \mathcal{F}) of assets. We define that using the auxiliary `eval` function, which maps configurations into sets of asset names— for a configuration c , the set that `eval` yields contain an asset name n iff there is a row in K that contains n and its expression evaluates to true according to c . We use the notation $A(_)$ for the relational image [30] of an asset mapping A , and $\exists i \in K \cdot p(i)$ as an abbreviation for the PVS notation $\exists i:\text{Item} \cdot i \in K \wedge p(i)$.

```

eval(K:CK, c:Configuration) :  $\mathcal{F}$ [AssetName] =
  {an |  $\exists i \in K \cdot \text{satisfies}(\text{exp}(i),c) \wedge \text{an} \in \text{assets}(i)$ }

semantics(K:CK, A:AM, c:Configuration) :  $\mathcal{F}$ [Asset] =
  A(eval(K,c))

```

To evaluate the CK, we need to check, for each item, if the feature expression is satisfied against the product configuration. We do this through the `satisfies` function. It evaluates a propositional formula against a configuration. For instance, feature expression $A \wedge B$ evaluates to true when we select both A and B in a product configuration. More importantly, a configuration c satisfies the feature name formula n if n is a value in c . For conciseness, we do not present here the complete PVS formalization, which is available in our online appendix [32].

3 Safe Evolution of Product Lines

In this section we introduce the necessary concepts about SPL refinement and CK equivalence to understand the algebraic laws presented in this work. We first present a formal definition for SPLs and the refinement notion previously defined [8]. We then present a novel CK equivalence notion, different than the one from previous works, as it is indexed by the FM, and it is necessary for some of the laws we propose.

To guide our SPL evolution analysis, we rely on the SPL refinement theory [8]. Such theory is based on an asset refinement notion, which is useful for comparing assets with respect to behavior preservation. For our purposes, an SPL consists of an FM, a CK, and an AM that jointly generate products, that is, well-formed asset sets in their target languages. Although we use the term FM, this theory is defined in a general way as to avoid depending on particular FM, CK, and asset languages. Thus, we do not depend on a particular FM notation, and could use alternatives such as decision models [36]. The theory just assumes a generic function, represented as $\llbracket F \rrbracket$, to obtain its semantics as a set of configurations. Hereafter, for making the discussion less abstract, we use FM terminology. The theory specifies AMs as a finite function from asset names to assets. Finally, it also abstracts the details for CK, representing its semantics as $\llbracket K \rrbracket_c^A$ —a function that receives a configuration knowledge K , an asset mapping A , and a product configuration c , to yield a finite set of assets that corresponds to a product from the SPL. The definition for SPLs given in what follows establishes that these three elements (FM, CK, AM) must jointly generate well-formed products. We use wf to represent the well-formedness constraint. It is necessary because missing an entry on a CK might lead to asset sets that do not correspond to valid products. Similarly, a mistake when writing a CK or AM entry might yield an invalid asset set due to conflicting assets. Since the theory does not depend on a particular asset language, wf might have different forms, depending on the particular asset language used. For instance, it might mean simply that the code compiles without errors.

Definition 1 (Product line)

For a feature model F , an asset mapping A , and a configuration knowledge K , we say that tuple (F, A, K) is a product line when, for all $c \in \llbracket F \rrbracket$, $wf(\llbracket K \rrbracket_c^A)$. \square

Similar to program refinement, SPL refinement is concerned with behavior preservation. However, it goes beyond code and other kinds of reusable assets, considering also FM and CK. In an SPL refinement, the resulting SPL should be able to generate products that behaviorally match the original SPL products. So users of an original product cannot observe behavior differences when using the corresponding product of the new SPL. This is exactly what guarantees safety when improving the SPL design.

In most SPL refinement scenarios, many changes need to be applied to code assets, FMs and CK, which often leads the refactored SPL to generate more

products than before [22]. As long as it generates enough products to match the original SPL, users have no reason to complain. We extend the SPL, not arbitrarily, but in a safe way. Figure 3, illustrates this by showing two SPLs, PL and PL' . In this example, PL is refined by PL' because for each product in PL (represented by a star), there is a corresponding product in PL' that refines it (represented by a square). As explained before, PL' can have new products and still preserve the refinement relation. This ensures that the transformation is safe; we extend the SPL without impacting existing users. We formalize these ideas in terms of asset set refinement. Basically, each product generated by the original SPL must be refined by some product of the new, improved, SPL.

Definition 2 (Product line refinement)

For product lines (F, A, K) and (F', A', K') , the second refines the first, denoted $(F, A, K) \sqsubseteq (F', A', K')$, whenever $\forall c \in \llbracket F \rrbracket \cdot \exists c' \in \llbracket F' \rrbracket \cdot \llbracket K \rrbracket_c^A \sqsubseteq \llbracket K' \rrbracket_{c'}^{A'}$. \square

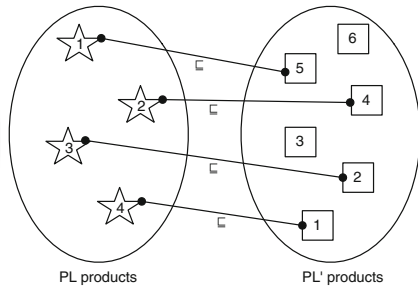


Fig. 3. Software product line refinement notion

Remember that, for a configuration c , a configuration knowledge K , and an asset mapping A related to a given SPL, $\llbracket K \rrbracket_c^A$ is a well-formed set of assets. Therefore, $\llbracket K \rrbracket_c^A \sqsubseteq \llbracket K' \rrbracket_{c'}^{A'}$ refers to asset set refinement. Again, as the general theory does not rely on a particular asset language, asset refinement is just assumed as a pre-order. The definition just mentioned relates two SPLs, therefore, all products in the target SPL should be well-formed. The \sqsubseteq symbol is used for SPLs as a relation that says when the transformation is safe. It does not mean functional equivalence because the refinement notion is a pre-order. It can reduce non-determinism, for example. The definition is also compositional, in the sense that refining an FM, AM, or CK that are part of a valid SPL yields a refined valid SPL. Such property is essential to guarantee independent development of these artifacts in an SPL. However, the CK equivalence previously defined [8] states that two CK models K and K' are equivalent, denoted $K \cong K'$, whenever $\llbracket K \rrbracket = \llbracket K' \rrbracket$. Notice that this definition demands equality of functions. The equivalence must hold for any asset mapping and configuration.

However, in some occasions, it is important to have a weaker CK equivalence definition, that is restricted by a particular FM, instead of demanding equality of

functions for any FM. In this work, we present a weaker equivalence notion, that is indexed by the FM, since the CK is used in conjunction with such artifact. For example, we can replace **On Demand** \vee **Startup** with **Image Load Policy** on the CK of Fig. 2. Even though these are not propositionally equivalent, we can do so because selecting **Image Load Policy** implies in selecting one of the two child features. We have proved this equivalence notion to be reflexive, symmetric, and transitive.

Definition 3 (Configuration Knowledge Equivalence)

For a feature model F , and configuration knowledge K and K' , we say that $K \cong_F K'$ when, for all asset mapping A and $c \in \llbracket F \rrbracket$, $\llbracket K \rrbracket_c^A = \llbracket K' \rrbracket_c^A$. \square

Since SPL refinement is reflexive [8], and the CK equivalence defined above ensures that all products remain the same, we can use this notion to justify safe evolution of SPLs when only the CK changes. The following theorem states this.

Theorem 1 (CK equivalence compositionality)

For product lines (F, A, K) and (F, A, K') , if $K \cong_F K'$ then $(F, A, K) \sqsubseteq (F, A, K')$. \square

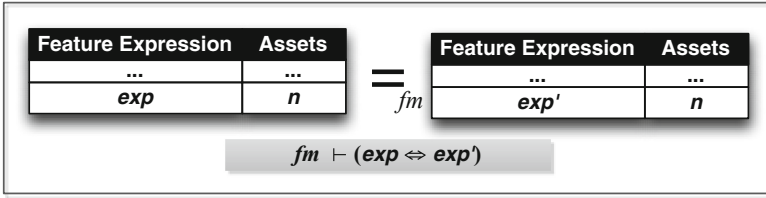
Based on these properties, we can propose transformations to avoid reasoning over the semantics of CK models, which might be complex, by specifying syntactic conditions where we can guarantee that the equivalence holds. If the transformations have conditions that depend on reasoning about the FM, we need to use the weaker equivalence notion. Otherwise, we can use the previously proposed stronger equivalence notion [8].

4 Configuration Knowledge Laws

In this section, we propose primitive laws, that is, transformations that fix some of the problems we mention in Sect. 2, while preserving the behavior of the SPL products [8]. Although primitive, we can compose them to derive coarse-grained transformations.

A law consists of two templates (patterns) of equivalent CK models, on the Left-Hand Side (LHS) and Right-Hand Side (RHS). For example, Law 1 establishes that we can replace a feature expression by another whenever their evaluation is equivalent by the FM. Therefore, we use the equivalence notion indexed by the FM, thus the fm symbol in the equality. A law may declare meta-variables. For instance, in Law 1, we use exp to denote a feature expression and n to denote an asset name. For simplicity we use a single asset name in the law, but the proof can handle any set of assets associated with exp . The dots represent other CK items that remain unaltered. We can specify a condition below the template. Since each law defines two semantics-preserving CK transformations, the condition holds for both directions of the transformations. The condition for Law 1 states that we can apply this law when, according to fm , exp is equivalent to exp' . A special case of this law is the case where expressions are equivalent by propositional reasoning only, and we would not need to check expressions against the FM.

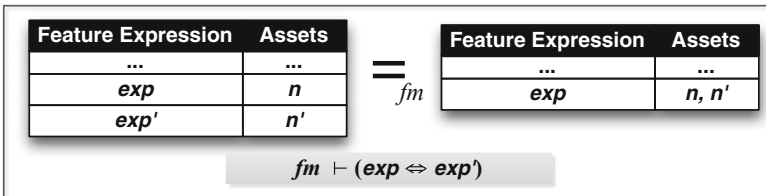
Law 1 *(Simplify Feature Expression)*



We apply a law whenever the CK matches a template and satisfy its conditions. A matching is an assignment of all variables occurring in LHS/RHS models to concrete values. There are occasions where it might be better to apply a law from left to right, whereas in other situation, it is better to apply it from right to left. Applying Law 1 can be useful to improve CK readability. It might also be useful to improve maintainability. For example, in Fig. 2, we can replace the feature expression **OnDemand** \vee **Startup** with **Image Loading Policy**, as they are equivalent by the FM, since both form an alternative group, therefore, whenever we select **Image Loading Policy**, we must select either **OnDemand** or **Startup**. If we add another image loading policy, we do not need to update the changed CK, while in the original, we would have to modify the feature expression.

Law 2 establishes that we can merge two CK items whenever their feature expressions are equivalent, according to the FM. We see that we merge *n* and *n'* into the same CK item, when applying the law from left to right. When applying from right to left, it also states that a CK item with multiple asset names can be splitted into items that have equivalent feature expressions. Application of this law from left to right can be useful in the context of improving long CK models, so we can reduce its size, thus, reducing cost for maintenance. However, it can also compromise readability.

Law 2 *(Merge items with equivalent feature expressions)*



Duplicated assets can happen in the CK. Law 3 states that we can merge CK items with duplicated assets. It establishes that, from left to right, we can merge two CK items into a single item, creating a new feature expression. The new expression is the disjunction of the previous feature expressions. Applying the law from right to left, we can split an *or* feature expression into CK items containing duplicated assets. Notice that there is no *fm* in the equality. Therefore, we use the stronger equivalence notion, which is not indexed by the FM, since we

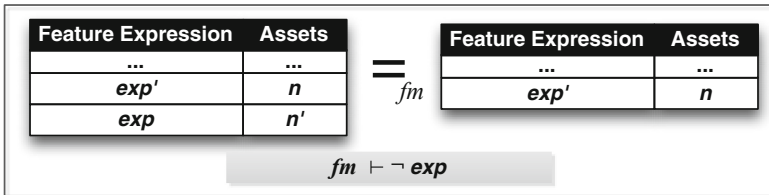
do not need it in this case. When applying from left to right, this law improves readability of CK models, since we avoid duplication of assets names in the CK.

Law 3 *(Duplicated Assets)*



A dead feature is a feature that does not appear in any product configuration from the FM [34]. During SPL evolution, modifications to the FM can result in dead feature expressions. That is, no product configurations can satisfy them. This might be the result of changing feature relationships. Law 4 establishes that we can remove a feature expression *exp* from the LHS CK, when we deduce from *fm* that it is never evaluated as true. Since we can represent FMs as logical propositions, we can efficiently check this [25]. This law also states that, when applying it from right to left, we can add a line with a dead feature expression to our CK, without changing the semantics.

Law 4 *(Dead Feature Expression)*



Law 5 states that the order in which we write the CK has no effect on its semantics. We can change the order of any item in the CK. This holds since we are dealing with assets selection only, not with complex transformations that might need some sort of ordering among them. This law is useful to generalize application of all other laws. Since we do not modify anything in the CK except for the order, it is straightforward to understand why this transformation preserves semantics. Evaluation of the CK for all product configurations yields the same set of assets in both LHS and RHS CK models.

Law 5 *⟨Change Order⟩*

5 Soundness

We also used PVS to prove all laws sound according to the CK equivalence notion (Definition 3). We use a similar approach as the one used for proving FM laws [16] and SPL refinement templates [8, 26, 28]. We structure each law as a theorem ensuring that it preserves the semantics of the CK, as follows. We represent the LHS and RHS CK models with the `ck1` and `ck2` variables, respectively. We represent FMs (`fm`) and AM (`am`) likewise. The `syntax` and `conditions` predicates describe syntactic similarities and differences between the LHS and RHS CK models in the law, and transformation conditions, respectively. The equivalence notion we use depends on the law we want to prove. When the law depends on the FM, we use the equivalence notion indexed by the FM (\cong_F). Otherwise, we use the stronger notion.

```
law: THEOREM
  ∀ fm:FM, ck1,ck2:CK ... ·
    syntax(...) ∧ conditions(...)
    ⇒ ck1 ≅ ck2
```

Next we specify these predicates for Law 1—Simplify Feature Expression. For each element in the template, we declare a variable for it in our PVS theory. For instance, the items which we refer to on Law 1 are represented by the PVS variables `it1` and `it2`, respectively. Aside from the item we are interested in changing, all other items from both CK models remain the same—`its`. For the item we are modifying, we are only changing the feature expression. Therefore, the asset set remains the same for `it1` and `it2`.

```
syntax(it1,it2:Item,its:ℙ[Item],ck1,ck2:CK):boolean =
  ck1 = {it1} ∪ its ∧ ck2 = {it2} ∪ its ∧ assets(it1) = assets(it2)
```

We specify the `conditions` predicate as follows. The `sat` function specifies that, for all product configurations that can be derived from `fm`, evaluation of the feature expression (`satisfies`) for both items must be equivalent.

```
conditions(it1,it2:Item,fm:FM) = sat(fm,exp(it1),exp(it2))
```

We now detail the proof for Law 1. For arbitrary fm , ck1 , ck2 , it1 and it2 , assume that the **syntax** and **conditions** predicates hold. Therefore, since we want to prove that $\text{ck1} \cong_{\text{fm}} \text{ck2}$ we need to prove, for an arbitrary am and a $\text{c} \in \text{semantics}(\text{fm})$, that

$$\text{semantics}(\text{ck1}, \text{am}, \text{c}) = \text{semantics}(\text{ck2}, \text{am}, \text{c})$$

Fully expanding the definition of semantics and replacing ck1 and ck2 with their respective values from the syntax predicate, we have to prove that

$$\begin{aligned} & \{ \text{a:Asset} \mid \\ & \quad \exists (\text{an:AssetName}) \cdot \\ & \quad \quad (\exists (\text{i:Item}) \cdot \text{item} = \text{it1} \vee \text{item} \in \text{its} \wedge \text{satisfies}(\text{exp}(\text{i}), \text{c}) \\ & \quad \quad \quad \wedge \text{an} \in \text{assets}(\text{i})) \wedge (\text{an}, \text{a}) \in \text{am} \} \\ & = \\ & \{ \text{a:Asset} \mid \\ & \quad \exists (\text{an:AssetName}) \cdot \\ & \quad \quad (\exists (\text{i:Item}) \cdot \text{item} = \text{it2} \vee \text{item} \in \text{its} \wedge \text{satisfies}(\text{exp}(\text{i}), \text{c}) \\ & \quad \quad \quad \wedge \text{an} \in \text{assets}(\text{i})) \wedge (\text{an}, \text{a}) \in \text{am} \} \end{aligned}$$

For items in its , it is straightforward to prove that the yielded assets are the same in both models. The only distinct items are it1 and it2 . Using the **conditions** predicate, we have that $\text{eval}(\text{exp}(\text{it1}), \text{c}) \Leftrightarrow \text{eval}(\text{exp}(\text{it2}), \text{c})$ for all $\text{c} \in \text{semantics}(\text{fm})$. This way, we are able to prove that the assets set yielded for ck1 is equal to the assets set yielded for ck2 , since we have that $\text{assets}(\text{it1}) = \text{assets}(\text{it2})$ in the **syntax** predicate.

Using similar reasoning, we can prove that Law 2 is sound, since we have the same condition for both laws. For Law 3, we prove soundness by demonstrating that the merging preserves semantics, since the RHS model yields a whether exp or exp' evaluates to true, according to a product configuration. Therefore, the assets set yielded remain the same as in the LHS model. Law 4 also preserves semantics, since we are introducing or removing an item whose feature expression is dead, meaning that it is not evaluated true for any product configuration. Therefore, it has no effect in the assets yielded when evaluating the CK. Finally, in Law 5, all items of both LHS and RHS CK models are the same, except for the order in which we write them. Therefore, it preserves semantics. Formal proofs and all PVS specification files are available at our online appendix [32].

6 Case Study

In this section, we analyze the evolution of a real SPL and evaluate how the laws can be used to evolve the CK during its development and maintenance. We analyzed TaRGeT [14], an SPL of tools that automatically generate functional tests from use case documents written in natural language. This SPL has been used by a mobile phone company to generate tests for its devices. It has six releases and over 32 KLOC on the latest. We manually analyzed CK transformations in

commits of its last three versioned releases, looking for scenarios where only the CK was changed. Moreover, after this evaluation, we have extended a toolset for checking SPL refinements [13] to include the automated checking for all of these laws.

We found four scenarios where we could justify safe evolution of the SPL applying the laws presented in this work. In the first one, the developer team replaced the feature expression $TC\ 3 \vee TC\ 4$ with **Output**, to improve CK readability and maintainability. Figure 4 depicts this transformation, presenting the CK before and after the transformation, as well as part of the FM that is of interest for this transformation, represented by fm in the equality. After performing this transformation, if we add a new child for **Output**, we do not need to update the feature expression in the CK anymore. We can justify safe evolution in this transformation applying Law 1, from left to right.

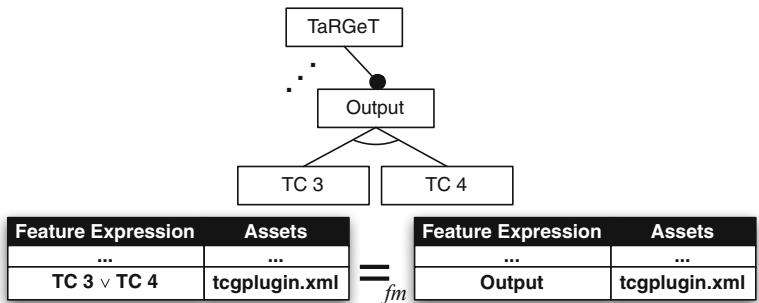


Fig. 4. Usage of Law 1 in TaRGeT SPL to simplify the feature expression.

In the second scenario, the developers merged the **TC 3** and **TC 4** feature expressions into a single item with the $TC\ 3 \vee TC\ 4$ expression, associated to the same asset. This improved CK readability, avoiding duplicated asset names. Figure 5 illustrates this transformation, with the CK models and part of the FM. We apply Law 3 from left to right to justify safe evolution.

In the third and fourth scenarios, developers merged duplicated feature expressions, to improve CK maintainability, reducing its size and removing duplicated feature expressions. Figure 6 presents the third scenario. They merged five items with the repeated **Interruption** expression into just one item and their associated assets. Similarly, Fig. 7 depicts the fourth transformation where they replaced repetitions of **Company 1** and **Company 2**. In both cases, we can use Law 2 from left to right to justify safe evolution. Additionally, in the fourth scenario, the developer team changed the order of the transformations, as presented in Law 5, to apply the previous transformation.

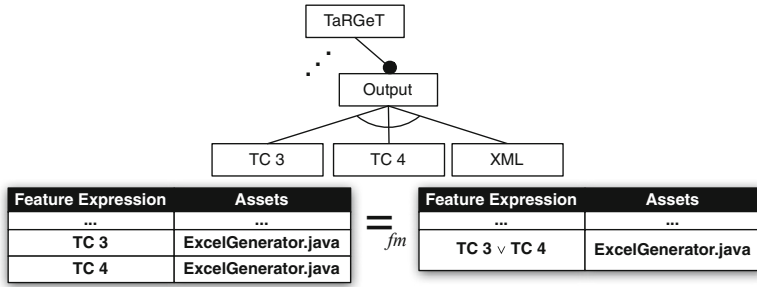


Fig. 5. Usage of Law 3 in TaRGeT SPL to remove assets duplication.

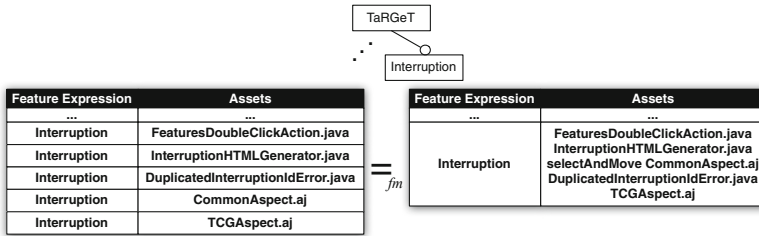


Fig. 6. Usage of Law 2 in TaRGeT SPL to remove duplicated expression.

The commits in TaRGeT consisted of many changes to different assets, instead of commits performed often, with small and localized changes. Therefore, it is possible that other scenarios where only the CK changed were not captured by the SPL version history. A manual analysis of 500 commits from the highly configurable system Soletta identified that 122 commits (24.4%) only change the Makefile [17], which would be analogous to the CK, for Kconfig-based systems. Moreover, 241 out of the 500 commits change the CK in conjunction with FM or code. Therefore, it is possible that changes to the CK are followed by changes to other SPL elements, and could benefit from the transformations proposed here.

7 Related Work

Several approaches [19, 20, 24, 35] focus on refactoring a single product into a SPL. Kolb et al. [20] discuss a case study in refactoring legacy code components into a product line implementation. They define a systematic process for refactoring products to obtain product lines assets. There is no discussion about feature models and configuration knowledge. Moreover, behavior preservation is only checked by testing. Similarly, Kästner et al. [19] focus only on transforming code assets, implicitly relying on refinement notions for aspect-oriented programs [10]. As discussed elsewhere [7] these are not adequate for justifying SPL refinement. Trujillo et al. [35] go beyond code assets, but do not explicitly consider CK transformations. They also do not consider behavior preservation;

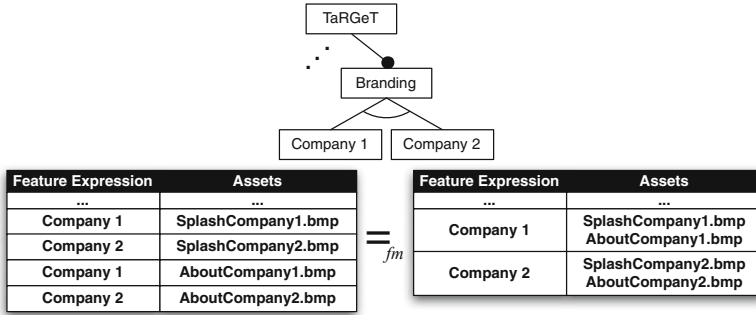


Fig. 7. Usage of Laws 2 and 5 in TaRGeT SPL to remove duplicated expressions.

they indeed use the term “refinement”, but in the different sense of overriding or adding extra behavior to assets. Moreover, all of these works focus on whole SPL evolution, while in this work, we focus on CK changes.

Liu et al. [24] also focus on the process of decomposing a legacy application into features, but go further than the previously cited approaches by proposing a refactoring theory that explains how a feature can be automatically associated to a code asset and related derivative assets, which contain feature declarations appropriate for different product configurations. Contrasting with our CK model, this theory assumes an implicit notion of CK based on the idea of derivatives. So it does not consider explicit CK transformations as we do here. In this work, we focus on refinements at a different level—models instead of programs.

Alves et al. extend the traditional notion of refactoring to SPLs [1]. Besides traditional program refactoring, they refactor FMs in order to improve configurability. They present and evaluate a set of sound FM refactorings in a real case study in the mobile domain. In an extension of this work, Gheyi et al. propose a sound, complete and minimal catalog of algebraic laws for FMs [16]. Moreover, they mechanized a theory for FMs in PVS, which we reuse in our theory for CK models in PVS. Nevertheless, these notions are not concerned with CK transformations, they only consider FM transformations.

An informal discussion of the CK semantics discussed here first appeared in a product line refactoring tutorial [7]. Besides talking about SPL and population refactoring, this tutorial illustrates different kinds of refactoring transformation templates that can be useful for deriving and evolving SPLs, including CK transformations. Later, the refinement theory was formalized [8], making clear the interface between the theory and languages used to describe SPL artifacts. Our work is complementary, since we use the CK language that instantiates the interface proposed, proposing individual CK transformations that adhere to the SPL refinement notion. Sampaio et al. extends the theory with a partial refinement notion, which formalizes the concept of partially safe evolution [28]—that is, evolution scenarios where only a subset of products from the original SPL has their behavior preserved. To use such notion we would need to propose transformations that are not behavior-preserving.

Recent works investigating the evolution characteristics of highly configurable systems show that evolving only the mapping between features and assets is not uncommon [12, 17, 21]. Kröher et al. evaluates the intensity of variability-related changes in the Linux kernel [21]. They measured how often changes occur in FM, AM, and CK, and how do those changes relate to variability information inside these artifacts. Our work could use theirs as input to identify potential instances where only the CK has been changed and investigate the use of the laws, if we adapted our model to comply with the one used by the Linux kernel. A tool with the same purpose is FEVER [12], which allows analysing Kconfig-based systems to extract feature-oriented change information. It is used in the work of Gomes et al. [17] to classifying evolution scenarios into safe or partially safe. Moreover, Gomes et al. also manually evaluated 500 commits from a system, showing that in 122 cases, only the CK has been changed by a commit. Again, this information provides evidence that changing only the CK is not uncommon.

8 Conclusions

In this work, we propose a set of laws for transforming CK models, that justify safe SPL evolution. We ensure this using the SPL refinement theory [8]. We propose a weaker equivalence notion and its associated compositionality result. We can apply these laws to fix problems such as duplicated assets and dead feature expressions. We use PVS to specify and prove soundness of the proposed laws. Therefore, developers do not need to reason based on semantics in order to refactor a CK, as the catalog can be directly applied. Thus, since SPL refinement is a pre-order, we can compose them with laws for FMs [16] and other SPL refinement templates [26] to derive elaborate transformations with the guarantee that we do not change behavior of existing products in the SPL.

As future work, we intend to investigate the completeness and minimality of our laws, proposing a CK normalization algorithm. Even though we did not achieve this result, we observed from our preliminary results, that the laws could justify safe evolution scenarios of a real SPL. We also intend to adapt our model and laws to consider KBuild notation, so we can evaluate it in other highly configurable systems that have been previously investigated in the context of safe and partially safe evolution [17, 28]. Moreover, we also intend to implement automated tool support for checking the conditions to apply the laws in this context. As the transformations precisely specify the mechanics and preconditions, their soundness is specially useful for correctly implementing the transformations and avoiding typical problems with current program refactoring tools [31]. Finally, the theory we present in this work formalizes part of the concepts and processes from tools [9, 24] and practical experience [1, 19, 20, 35] on SPL refactoring.

Acknowledgments. This work was partially supported by CAPES (grants 117875 and 175956), CNPq (grants 409335/2016-9, 426005/2018-0, and 311442/2019-6) and FACEPE (APQ-0570-1.03/14), as well as INES 2.0, (<http://www.ines.org.br>) FACEPE grants PRONEX APQ-0388-1.03/14 and APQ-0399-1.03/17, and CNPq grant 465614/2014-0.

References

1. Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., Lucena, C.: Refactoring product lines. In: GPCE 2006, pp. 201–210 (2006)
2. Apel, S., Batory, D., Kstner, C., Saake, G.: Feature-Oriented Software Product-Lines: Concepts and Implementation. Springer, Heidelberg (2013). <https://doi.org/10.1007/978-3-642-37521-7>
3. Batory, D.S.: Feature-oriented programming and the AHEAD tool suite. In: ICSE 2004, pp. 702–703 (2004)
4. Batory, D.: Feature models, grammars, and propositional formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005). https://doi.org/10.1007/11554844_3
5. Berger, T., She, S., Lotufo, R., Wasowski, A., Czarnecki, K.: Variability modeling in the real: a perspective from the operating systems domain. In: ASE 2010, pp. 73–82 (2010)
6. Bonifácio, R., Borba, P.: Modeling scenario variability as crosscutting mechanisms. In: AOSD 2009, pp. 125–136 (2009)
7. Borba, P.: An introduction to software product line refactoring. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2009. LNCS, vol. 6491, pp. 1–26. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18023-1_1
8. Borba, P., Teixeira, L., Gheyi, R.: A theory of software product line refinement. *Theoret. Comput. Sci.* **455**, 2–30 (2012)
9. Calheiros, F., Borba, P., Soares, S., Nepomuceno, V., Alves, V.: Product line variability refactoring tool. In: WRT 2007 at ECOOP 2007, pp. 33–34 (2007)
10. Cole, L., Borba, P.: Deriving refactorings for AspectJ. In: AOSD 2005, pp. 123–134. ACM (2005)
11. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. ACM Press/Addison-Wesley, New York (2000)
12. Dintzner, N., van Deursen, A., Pinzger, M.: FEVER: an approach to analyze feature-oriented changes and artefact co-evolution in highly configurable systems. *Empirical Softw. Eng.* **23**(2), 905–952 (2017). <https://doi.org/10.1007/s10664-017-9557-6>
13. Ferreira, F., Gheyi, R., Borba, P., Soares, G.: A toolset for checking SPL refinements. *J. Univ. Comput. Sci.* **20**(5), 587–614 (2014)
14. Ferreira, F., Neves, L., Silva, M., Borba, P.: Target: a model based product line testing tool. In: Tools Session at CBSOFT 2010, pp. 67–72 (2010)
15. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, New York (1999)
16. Gheyi, R., Massoni, T., Borba, P.: Algebraic laws for feature models. *J. Univ. Comput. Sci.* **14**(21), 3573–3591 (2008)
17. Gomes, K., Teixeira, L., Alves, T., Ribeiro, M., Gheyi, R.: Characterizing safe and partially safe evolution scenarios in product lines: An empirical study. In: VaMoS 2019 (2019)
18. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Technical report, CMU Software Engineering Institute (1990)
19. Kastner, C., Apel, S., Batory, D.: A case study implementing features using AspectJ. In: SPLC 2007, pp. 223–232. IEEE (2007)
20. Kolb, R., Muthig, D., Patzke, T., Yamauchi, K.: A case study in refactoring a legacy component for reuse in a product line. In: ICSM 2005, pp. 369–378 (2005)

21. Kröher, C., Gerling, L., Schmid, K.: Identifying the intensity of variability changes in software product line evolution. In: SPLC, pp. 54–64 (2018)
22. Krueger, C.W.: Easing the transition to software mass customization. In: van der Linden, F. (ed.) PFE 2001. LNCS, vol. 2290, pp. 282–293. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-47833-7_25
23. Van der Linden, F., Schmid, K., Rommes, E.: Software Product Lines in Action: the Best Industrial Practice in Product Line Engineering. Springer, Heidelberg (2007). <https://doi.org/10.1007/978-3-540-71437-8>
24. Liu, J., Batory, D., Lengauer, C.: Feature oriented refactoring of legacy applications. In: ICSE 2006, pp. 112–121 (2006)
25. Mendonca, M., Wasowski, A., Czarnecki, K.: Sat-based analysis of feature models is easy. In: SPLC 2009, pp. 231–240 (2009)
26. Neves, L., Borba, P., Alves, V., Turnes, L., Teixeira, L., Sena, D., Kulesza, U.: Safe evolution templates for software product lines. JSS **106**(C), 42–58 (2015)
27. Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS Language Reference. SRI International (2001). <http://pvs.csl.sri.com/doc/pvs-language-reference.pdf>, version 2.4
28. Sampaio, G., Borba, P., Teixeira, L.: Partially safe evolution of software product lines. J. Syst. Softw. **155**, 17–42 (2019)
29. Schobbens, P.Y., Heymans, P., Trigaux, J.C., Bontemps, Y.: Generic semantics of feature diagrams. Comput. Netw. **51**(2), 456–479 (2007)
30. Spivey: The Z Notation: A Reference Manual. Prentice Hall (1987)
31. Steimann, F., Thies, A.: From public to private to absent: refactoring JAVA programs under constrained accessibility. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 419–443. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03013-0_19
32. Teixeira, L., Gheyi, R., Borba, P.: Online appendix (2020). <http://www.cin.ufpe.br/~lmt/sbmf2020/>
33. Teixeira, L., Alves, V., Borba, P., Gheyi, R.: A product line of theories for reasoning about safe evolution of product lines. In: SPLC 2015, pp. 161–170 (2015)
34. Trinidad, P., Benavides, D., Durán, A., Cortés, A.R., Toro, M.: Automated error analysis for the agilization of feature modeling. JSS **81**(6), 883–896 (2008)
35. Trujillo, S., Batory, D., Diaz, O.: Feature refactoring a multi-representation program into a product line. In: GPCE 2006, pp. 191–200 (2006)
36. Weiss, D.M., Li, J.J., Slye, J.H., Dinh-Trong, T.T., Sun, H.: Decision-model-based code generation for SPLE. In: SPLC 2008, pp. 129–138 (2008)