A Theory of Software Product Line Refinement

Paulo Borba, Leopoldo Teixeira and Rohit Gheyi

Informatics Center – Federal University of Pernambuco
Department of Computing Systems – Federal University of Campina Grande
{phmb,lmt}@cin.ufpe.br -- rohit@dsc.ufcg.edu.br

Abstract. To safely derive and evolve a software product line, it is important to have a notion of product line refactoring and its underlying refinement notion, which assures behavior preservation. In this paper we present a general theory of product line refinement by extending a previous formalization with explicit interfaces between our theory and the different languages that can be used to create product line artifacts. More important, we establish product line refinement properties that justify stepwise and compositional product line development and evolution.

1 Introduction

A software product line is a set of related software products that are generated from reusable assets. Products are related in the sense that they share common functionality. Assets correspond to components, classes, property files, and other artifacts that are composed in different ways to specify or build the different products. This kind of reuse targeted at a specific set of products can bring significant productivity and time to market improvements [PBvdL05,vdLSR07].

To obtain these benefits with reduced upfront investment, previous work [Kru02,CN01,AJC+05] proposes to minimize the initial product line (domain) analysis and development process by bootstraping existing related products into a product line. In this context it is important to rely on a notion of product line refactoring [Bor09], which provides guidance and safety for deriving a product line from existing products, and also for evolving a product line by simply improving its design or by adding new products while preserving existing ones. Product line refactoring goes beyond program refactoring notions [Opd92,Fow99,BSCC04,CB05] by considering both sets of reusable assets that not necessarily correspond to valid programs, and extra artifacts, such as feature models [KCH+90,CE00], which are necessary for automatically generating products from assets.

Instead of focusing on the stronger notion of refactoring, in this paper we focus on the underlying notion of product line refinement, which

also captures behavior preservation but abstracts quality improvement. This allows us to develop a formal theory of product line refinement, extending the previous formalization [Bor09] with explicit assumptions about the different languages that can be used to create product line artifacts. More important, we establish product line refinement properties that justify safe stepwise and compositional product line development and evolution. Our theory is encoded in the Prototype Verification System (PVS) [ORS92], which provides mechanized support for formal specification and verification. All properties are proved using the PVS prover.

This text is organized as follows. Section 2 introduces basic concepts and notation for feature models and other extra product line artifacts [CE00,BB09]. Several assumptions and axioms explicitly establish the interfaces between our theory and particular languages used to describe a product line. Definitions and lemmas are introduced to formalize auxiliary concepts and properties. Following that, in Sec. 3, we discuss and formalize our notion of product line refinement. We also derive basic properties that justify stepwise product line development and evolution. Next, Sec. 4 presents the product line refinement compositionality results and their proofs. We discuss related work in Sec. 5 and conclude with Sec. 6. Finally, Appendix A contains proofs omitted in the main text.

2 Product lines concepts

In the product line approach formalized in this paper, automatic generation of products from assets is enabled by Feature Models and Configuration Knowledge (CK) [CE00]. A feature model specifies common and variant features among products, and is used for describing and selecting products based on the features they support. A CK relates features and assets, specifying which assets implement possible feature combinations. Hence a CK can be used to actually build a product given chosen features for that product. We now explain in more detail these two kinds of artifacts and related concepts, using examples from the Mobile Media product line [FCS+08], which contains applications – such as the one illustrated in Fig. 1 – that manipulate photo, music, and video on mobile devices.

2.1 Feature models

A feature model is essentially represented as a tree, containing features and information about how they are related. Features basically abstract

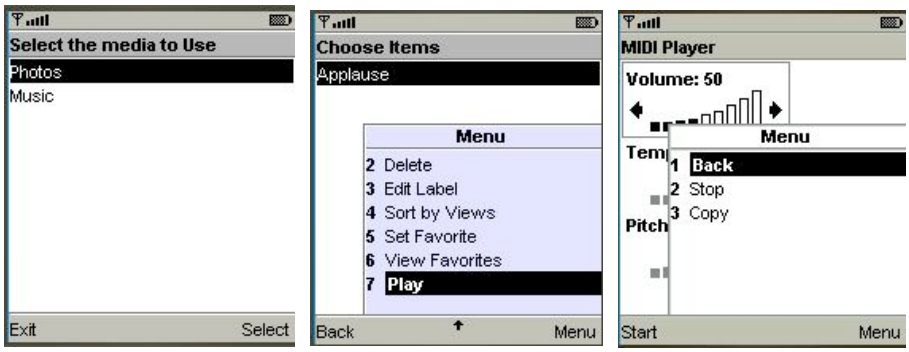

Fig. 1. Mobile Media screenshots

groups of associated requirements, both functional and non-functional. In the particular feature model notation illustrated here, relationships between a parent feature and its child features (subfeatures) indicate whether the subfeatures are *optional* (present in some products but not in others, represented by an unfilled circle), *mandatory* (present in all products, represented by a filled circle), *or* (every product has at least one of them, represented by a filled triangular shape), or *alternative* (every product has exactly one of them, represented by an unfilled triangular shape). For example, Fig. 2 depicts a simplified Mobile Media feature model, where Sorting is optional, Media is mandatory, Photo and Music are or-features, and the two illustrated screen sizes are alternative.

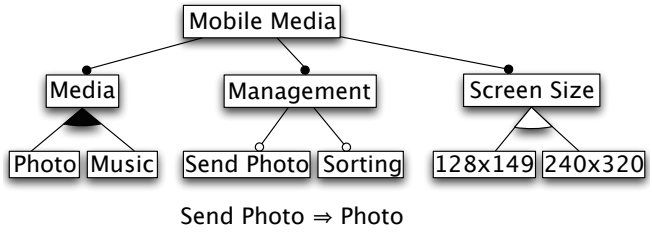

Fig. 2. Mobile Media simplified feature model

Besides these relationships, feature models may contain propositional logic formulas about features. Feature names are used as atoms to indicate that a feature should be selected. So negation of a feature indicates that

it should not be selected. For instance, the formula just below the tree in Fig. 2 states that feature Photo must be present in some product whenever feature Send Photo is selected. So

{Photo, Send Photo, 240x320},

together with the mandatory features, which hereafter we omit for brevity, is a valid feature selection (product configuration), but

{Music, Send Photo, 240x320}

is not. Likewise {Music, Photo, 240x320} is a valid configuration, but

{Music, Photo, 240x320, 128x149}

is not because it breaks the Screen Size alternative constraint. In summary, a valid configuration is one that satisfies all feature model constraints, specified both graphically and through formulas.

The set of all valid configurations often represents the semantics of a feature model. However, as different feature model notations might express constraints and configurations in different ways, our product line refinement theory abstracts the details and just assumes a generic function $[\![\_]\!]$ for obtaining the semantics of a feature model as a set of configurations.

Assumption 1 ⟨Feature model semantics⟩

$FeatureModel : TYPE$

$Configuration : TYPE$

$[\![\_]\!] : FeatureModel \rightarrow set[Configuration]$

We use simplified PVS notation for introducing the mentioned function and related types. In PVS, $TYPE$ declares an uninterpreted type that imposes no assumptions on implementations of the specification.

As shall be clear latter, these concepts are all we require about feature models. With them, we can define our product line refinement notion and derive its properties. So our theory applies for any feature model notation whose semantics can be expressed as a set of configurations. This is the case of the feature model notation illustrated in this section and others, which have been formalized elsewhere [GMB08,AGM+06,CHE05,Bat05,SHTB07].

Given a notion of feature model semantics, it is useful to define a notion of feature model equivalence to reason about feature models. Two feature models are equivalent iff they have the same semantics.

Definition 1 ⟨Feature model equivalence⟩
Feature models F and F' are equivalent, denoted $F \cong F'$, whenever $[\![F]\!] = [\![F']\!]$.

Again, this is quite similar to the PVS specification, which defines the equivalence as a function with the following type:

$$\cong : FeatureModel, FeatureModel \rightarrow bool$$

Hereafter we omit such typing details, and overload symbols, but the types can be easily inferred from the context.

We now establish the equivalence properties for the just introduced function.

Theorem 1 ⟨Feature model equivalence – reflexivity⟩

$$\forall F : FeatureModel \cdot F \cong F$$

Proof: Follows directly from Definition 1 and the reflexivity of the equality of configuration sets. □

Theorem 2 ⟨Feature model equivalence – symmetry⟩

$$\forall F, F' : FeatureModel \cdot F \cong F' \Rightarrow F' \cong F$$

Proof: Follows directly from Definition 1 and the symmetry of the equality of configuration sets. □

Theorem 3 ⟨Feature model equivalence – transitivity⟩

$$\forall F, F', F'' : FeatureModel \cdot F \cong F' \wedge F' \cong F'' \Rightarrow F \cong F''$$

Proof: Follows directly from Definition 1 and the transitivity of the equality of configuration sets. □

These properties justify safe stepwise evolution of feature models, as illustrated in previous work [AGM+06].

2.2 Assets and products

Besides a precise notion of feature model semantics, for defining product line refinement we assume means of comparing assets and products with respect to behavior preservation. We distinguish arbitrary asset sets ($set[Asset]$) from well-formed asset sets ($Product$), which correspond to valid products in the underlying languages used to describe assets. We assume the wf function specifies well-formedness, and $\sqsubseteq$ denotes both asset and product refinement.

Assumption 2 ⟨Asset and product refinement⟩

$$Asset : TYPE$$
$$\sqsubseteq : Asset, Asset \rightarrow bool$$
$$wf : set[Asset] \rightarrow bool$$
$$Product : TYPE = (wf)$$
$$\sqsubseteq : Product, Product \rightarrow bool$$

We use the PVS notation for defining the *Product* type as the set of all asset sets that satisfy the wf predicate.

Our product line refinement theory applies for any asset language with these notions as long as they satisfy the following properties. Both asset and product refinement must be pre-orders.

Axiom 1 ⟨Asset refinement reflexivity⟩

$$\forall a : Asset \cdot a \sqsubseteq a$$

Axiom 2 ⟨Asset refinement transitivity⟩

$$\forall a, b, c : Asset \cdot a \sqsubseteq b \wedge b \sqsubseteq c \Rightarrow a \sqsubseteq c$$

Axiom 3 ⟨Product refinement reflexivity⟩

$$\forall p : Product \cdot p \sqsubseteq p$$

Axiom 4 ⟨Product refinement transitivity⟩

$$\forall p, q, r : Product \cdot p \sqsubseteq q \wedge q \sqsubseteq r \Rightarrow p \sqsubseteq r$$

These are usually properties of any refinement notion because they are essential to support stepwise refinement and development. This is, for example, the case of existing refinement notions for object-oriented programming and modeling [BSCC04,GMB05,MGB08].

Finally, asset refinement must be compositional in the sense that refining an asset that is part of a valid product yields a refined valid product.

Axiom 5 ⟨Asset refinement compositionality⟩

$$\begin{aligned}&\forall a, a' : Asset \cdot \forall s : set[Asset] \cdot \\ &\qquad a \sqsubseteq a' \wedge wf(a \cup s) \\ &\quad \Rightarrow wf(a' \cup s) \wedge a \cup s \sqsubseteq a' \cup s\end{aligned}$$

We use $\cup$ both to denote set union and insertion of an element to a set.

Such a compositionality property is essential to guarantee independent development of assets in a product line, and is supported, for example, by existing class refinement notions [SB04]. In that context, a product is a main command with a set of class declarations that coherently resolves all references to class and method names. In general, we do not have to limit ourselves to code assets, and consider any kind of asset that supports the concepts and properties discussed in this section.

2.3 Configuration knowledge

As discussed in Sec. 2.1, features are groups of requirements, so they must be related to the assets that realize them. This is specified by the configuration knowledge (CK), which can be expressed in many ways, including as a relation from feature expressions (propositional formulas having feature names as atoms) to sets of asset names [BB09]. For example, showing the relation in tabular form, the following CK

Mobile Media	MM.java, ...
Photo	Photo.java, ...
Music	Music.java, ...
Photo $\vee$ Music	Common.aj, ...
Photo $\wedge$ Music	AppMenu.aj, ...
$\vdots$	$\vdots$

establishes that if the Photo and Music features are both selected then the `AppMenu` asset, among others omitted in the fifth row, should be part of the final product. Essentially, this product line uses the `AppMenu` aspect as a variability implementation mechanism [GA01,AJC+05] that has the effect of presenting the left screenshot in Fig. 1. For usability issues, this screen should not be presented by products that have only one of the Media features, so the need for the fifth row in the simplified Mobile Media CK. Similarly, some assets are shared by the Photo and Music implementations, so we write the fourth row to avoid repeating the asset names on the second and third rows.

Given a valid product configuration, the evaluation of a CK yields the names of the assets needed to build the corresponding product. In our example, the configuration {Photo, 240x320}[1] leads to

{`MM.java`, ..., `Photo.java`, ..., `Commom.aj`, ...}.

[1] Remember we omit mandatory features for brevity.

This gives the basic intuition for the semantics of a CK. It is a function that maps product configurations into finite sets (represented by *fset*) of asset names. So our product line refinement theory relies on a CK semantic function $[\![\_]\!]$ as follows.

Assumption 3 ⟨CK semantics⟩

$$CK : TYPE$$
$$AssetName : TYPE$$
$$[\![\_]\!] : CK \rightarrow Configuration \rightarrow fset[AssetName]$$

For the CK notation illustrated in this section, the semantics of a given CK K, represented as $[\![K]\!]$, could be defined in the following way: for a configuration c, an asset name n is in the set $[\![K]\!]c$ iff there is a row in K that contains n and its expression evaluates to true according to c. But we do not give further details because our aim is to establish a product line refinement theory that is independent of CK notation, as long as this notation's semantics can be expressed as a function that maps configurations into finite sets of assets names.

Similarly to what we have done for feature models, we define a notion of CK equivalence based on the notion of CK semantics. This is useful to reason about CK. Two CK specifications are equivalent iff they have the same semantics.

Definition 2 ⟨Configuration knowledge equivalence⟩
Configuration knowledge K is equivalent to K', denoted $K \cong K'$, whenever $[\![K]\!] = [\![K']\!]$.

We now establish the equivalence properties for the just introduced relation.

Theorem 4 ⟨Configuration knowledge equivalence – reflexivity⟩

$$\forall K : CK \cdot K \cong K$$

Proof: Follows directly from Definition 2 and the reflexivity of the equality of functions. □

Theorem 5 ⟨Configuration knowledge equivalence – symmetry⟩

$$\forall K, K' : CK \cdot K \cong K' \Rightarrow K' \cong K$$

Proof: Follows directly from Definition 2 and the symmetry of the equality of functions. □

Theorem 6 ⟨Configuration knowledge equivalence – transitivity⟩

$$\forall K, K', K'' : CK \cdot K \cong K' \wedge K' \cong K'' \Rightarrow K \cong K''$$

Proof: Follows directly from Definition 2 and the transitivity of the equality of functions.

□

2.4 Asset mapping

Although the CK illustrated in the previous section refers only to code assets, in general we could also refer to requirements documents, design models, test cases, image files, XML files, and so on. For simplicity, we focus on code assets as they are equivalent to other kinds of asset for our purposes. The important issue here is not the nature of asset contents, but how the assets are compared and referred to in the CK.

We cover asset comparison in Sec. 2.2. For dealing with asset references, each product line keeps a mapping such as the following

$$\{\text{Main 1} \mapsto \texttt{class Main \{ ...new StartUp(...);... \}},\ \text{Main 2} \mapsto \texttt{class Main \{ ...new OnDemand(...);... \}},\ \text{Common.java} \mapsto \texttt{class Common \{ ... \}} \vdots\ \}$$

from asset names used in a CK to actual assets. So, besides a feature model and a CK, a product line contains an asset mapping, which basically corresponds to an environment of asset declarations. This allows conflicting assets in a product line, like assets that implement alternative features, such as both `Main` classes in the illustrated asset mapping.

Formally, we specify asset mappings in PVS as follows.

Definition 3 ⟨Asset mapping⟩
Let r be a finite set of name-asset pairs ($r : fset[AssetName, Asset]$).

$$
\begin{aligned}
&mapping(r) : bool = \\
&\quad \forall n : AssetName \cdot \forall a, b : Asset \cdot \\
&\qquad (n, a) \in r \wedge (n, b) \in r \Rightarrow a = b \\
&AssetMapping : TYPE = (mapping)
\end{aligned}
$$

Since there is not much to abstract from this notion of asset mapping, it is actually defined as part of our theory. Differently from the concepts of feature model, CK, and their semantics, the asset mapping concept is not a parameter to our theory.

We also define auxiliary functions that are used to define product line refinement. The second one is mapping application over a set. In the following, consider that $m : AssetMapping$ and $s : fset[AssetName]$.

Definition 4 ⟨Auxiliary asset mapping functions⟩

$$
\begin{aligned}
&dom(m) : set[AssetName] = \\
&\quad \{n : AssetName \mid \exists a : Asset \cdot (n, a) \in m\} \\
&m\langle s \rangle : set[Asset] = \\
&\quad \{a : Asset \mid \exists n \in s \cdot (n, a) \in m\}
\end{aligned}
$$

We use the notation $\exists n \in s \cdot p(n)$ as an abbreviation for the PVS notation $\exists n : AssetName \cdot n \in s \wedge p(n)$.

To derive product line refinement properties, we establish several properties of the introduced auxiliary functions. The proofs appear in Appendix A.

Lemma 1 ⟨Distributed mapping over union⟩
For asset mapping A, asset a, and finite sets of asset names S and S', if

$$a \in A\langle S \cup S' \rangle$$

then

$$a \in A\langle S \rangle \vee a \in A\langle S' \rangle$$

□

Lemma 2 ⟨Distributed mapping over singleton⟩
For asset mapping A, asset name an and finite set of asset names S, if

$$an \in dom(A)$$

then

$$\exists a : Asset \cdot (an, a) \in A \ \wedge \ A\langle an \cup S\rangle = a \cup A\langle S\rangle$$

□

Remember we use $\cup$ both for set union and insertion of an element to a set.

Lemma 3 ⟨Asset mapping domain membership⟩
For asset mapping A, asset name an and asset a, if

$$(an, a) \in A$$

then

$$an \in dom(A)$$

□

Lemma 4 ⟨Distributed mapping over set of non domain elements⟩
For asset mapping A and finite set of asset names S, if

$$\neg\exists n \in S \cdot n \in dom(A)$$

then

$$A\langle S\rangle = \{\}$$

□

For reasoning about asset mappings, we define a notion of asset mapping refinement. Asset mapping equivalence could also be defined, but we choose the weaker refinement notion since it gives us more flexibility when evolving asset mappings independently of other product line elements such as feature models and CK. As shall be clear latter, we can rely on refinement for asset mappings but not for the other elements; that is why, in previous sections, we define equivalences for them. For asset mapping refinement, exactly the same names should be mapped, not necessarily to the same assets, but to assets that refine the original ones.

Definition 5 ⟨Asset mapping refinement⟩
For asset mappings A and A', the first is refined by the second, denoted

$$A \sqsubseteq A'$$

whenever

$$\begin{array}{l} \quad dom(A) = dom(A') \\ \wedge\, \forall n \in dom(A) \cdot \\ \quad\quad \exists a, a' : Asset \cdot (n, a) \in A \wedge (n, a') \in A' \wedge a \sqsubseteq a' \end{array}$$

We use $\forall n \in dom(A) \cdot p(n)$ to abbreviate the PVS notation

$$\forall n : AssetName \cdot n \in dom(A) \Rightarrow p(n)$$

Note also that $a \sqsubseteq a'$ in the definition refers to asset refinement, not to program refinement.

We now prove that asset mapping refinement is a pre-order.

Theorem 7 ⟨Asset mapping refinement reflexivity⟩

$$\forall A : AssetMapping \cdot A \sqsubseteq A$$

Proof: For an arbitrary asset mapping A, from Definition 5 we have to prove that

$$\begin{array}{l} \quad dom(A) = dom(A) \\ \wedge\, \forall n \in dom(A) \cdot \\ \quad\quad \exists a, a' : Asset \cdot (n, a) \in A \wedge (n, a') \in A \wedge a \sqsubseteq a' \end{array}$$

The first part of the conjunction follows from equality reflexivity. For an arbitrary $n \in dom(A)$, we are left to prove

$$\exists a, a' : Asset \cdot (n, a) \in A \wedge (n, a') \in A \wedge a \sqsubseteq a' \tag{1}$$

From Definition 4, as $n \in dom(A)$, we have that

$$n \in \{n : AssetName \mid \exists a : Asset \cdot (n, a) \in A\}$$

By set comprehension and membership, we have that

$$\exists a : Asset \cdot (n, a) \in A$$

Let a_1 be such a. Then we have $(n, a_1) \in A$. From this and Axiom 1, we easily obtain 1 taking a and a' as a_1.

□

Theorem 8 ⟨Asset mapping refinement transitivity⟩

$$\forall A, A', A'' : AssetMapping \cdot A \sqsubseteq A' \wedge A' \sqsubseteq A'' \Rightarrow A \sqsubseteq A''$$

Proof: For arbitrary asset mappings A, A', and A'', assume that $A \sqsubseteq A'$ and $A' \sqsubseteq A''$. From Definition 5 we have to prove that

$$\begin{aligned} & dom(A) = dom(A'') \\ \wedge\ & \forall n \in dom(A) \cdot \\ & \quad \exists a, a'' : Asset \cdot (n, a) \in A \wedge (n, a'') \in A'' \wedge a \sqsubseteq a'' \end{aligned}$$

The first part of the conjunction follows from our assumptions, Definition 5, and equality transitivity. For an arbitrary $n \in dom(A)$, we are left to prove

$$\exists a, a'' : Asset \cdot (n, a) \in A \wedge (n, a'') \in A'' \wedge a \sqsubseteq a'' \tag{2}$$

But from our assumptions and Definition 5 we have that $n \in dom(A')$ and therefore

$$\begin{aligned} & (n, a) \in A \wedge (n, a') \in A' \wedge a \sqsubseteq a' \\ & (n, a') \in A' \wedge (n, a'') \in A'' \wedge a' \sqsubseteq a'' \end{aligned}$$

for some $a, a', a'' : Asset$. We then have the a and a'' necessary to obtain 2 directly from this and the transitivity of asset refinement (Axiom 2). □

To establish the compositionality results, we rely on an important property of asset mapping refinement: if $A \sqsubseteq A'$ then products formed by using A assets are refined by products formed by corresponding A' assets.

Lemma 5 ⟨Asset mapping compositionality⟩
For asset mapping A and A', if

$$A \sqsubseteq\ A'$$

then

$$\begin{aligned} \forall ans : fset[AssetName] \cdot \forall as : fset[Asset] \cdot & \\ & wf(as \cup A\langle ans \rangle) \\ \Rightarrow\ & wf(as \cup A'\langle ans \rangle) \wedge as \cup A\langle ans \rangle \sqsubseteq as \cup A'\langle ans \rangle \end{aligned}$$

□

2.5 Product lines

We can now provide a precise definition for product lines. In particular, a product line consists of a feature model, a CK, and an asset mapping that jointly generate products, that is, valid asset sets in their target languages.

Definition 6 ⟨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 $c \in [\![F]\!]$,

$$wf(A\langle [\![K]\!]c\rangle)$$

We omit the PVS notation for introducing the *ProductLine* type, but it roughly corresponds to the one we use in this definition.

The well-formedness constraint in the definition is necessary because missing an entry on a CK might lead to asset sets that are missing some parts and thus are not valid products. Similarly, a mistake when writing a CK or asset mapping entry might yield an invalid asset set due to conflicting assets, like two aspects that are used as variability mechanism [GA01,AJC+05] and introduce methods with the same signature in the same class. Here we demand product line elements to be coherent as explained.

Given the importance of the well-formedness property in this definition, we establish compositionality properties related to the well-formedness function wf. First we have that feature model equivalence is compositional with respect to wf.

Lemma 6 ⟨Feature model equivalence compositionality over wf⟩
For feature models F and F', asset mapping A, and configuration knowledge K, if

$$F \cong F' \wedge \forall c \in [\![F]\!] \cdot wf(A\langle [\![K]\!]c\rangle)$$

then

$$\forall c \in [\![F']\!] \cdot wf(A\langle [\![K]\!]c\rangle)$$

□

Similarly, for CK we have the following.

Lemma 7 ⟨CK equivalence compositionality over wf⟩
For feature model F, asset mapping A, and configuration knowledge K and K', if

$$K \cong K' \wedge \forall c \in [\![F]\!] \cdot wf(A\langle [\![K]\!]c\rangle)$$

then

$$\forall c \in [\![F]\!] \cdot wf(A\langle [\![K']\!]c\rangle)$$

□

Finally, for asset mappings we have that refinement is compositional with respect to wf.

Lemma 8 ⟨Asset mapping refinement compositionality over wf⟩
For feature model F, asset mapping A and A' and configuration knowledge K, if

$$A \sqsubseteq A' \wedge \forall c \in [\![F]\!] \cdot wf(A\langle [\![K]\!]c\rangle)$$

then

$$\forall c \in [\![F]\!] \cdot wf(A'\langle [\![K]\!]c\rangle)$$

□

3 Product line refinement

Now that we better understand what a product line is, we can introduce a notion of product line refinement that provides guidance and safety for deriving a product line from existing products, and also for evolving a product line by simply improving its design or by adding new products while preserving existing ones.

Similar to program and model refinement [BSCC04,GMB05], product line refinement preserves behavior. However, it goes beyond source code and other kinds of reusable assets, and considers transformations to feature models and CK as well. This is illustrated by Fig. 3, where we refine the simplified Mobile Media product line by renaming the feature Music.

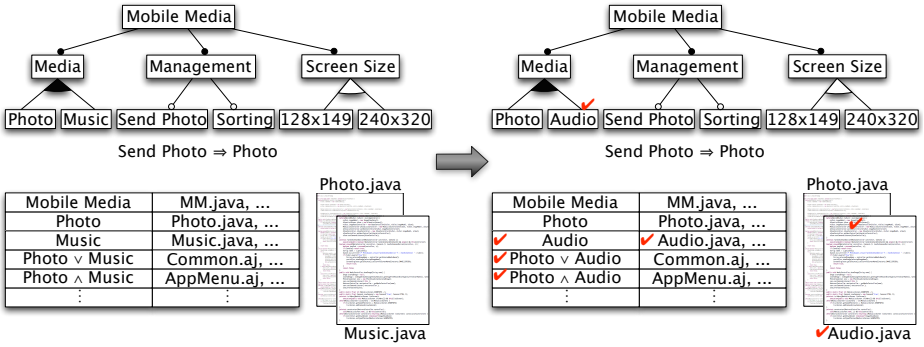

Fig. 3. Product line renaming refinement

As indicated by check marks, this renaming requires changing the feature model, CK, and asset mapping; due to a class name change, we must apply a global renaming, so the main method and other classes beyond `Music.java` are changed too.

The notion of behavior preservation should be also lifted from assets to product lines. In a product line refinement, the resulting product line should be able to generate products that behaviorally match the original product line products. So users of an original product cannot observe behavior differences when using the corresponding product of the new product line. With the renaming refinement, for example, we have only improved the product line design: the resulting product line generates a set of products exactly equivalent to the original set. But it should not be always like that. We consider that the better product line might generate more products than the original one. As long as it generates enough products to match the original product line, users have no reason to complain. For instance, by adding the optional Copy feature (see Fig. 4), we refine our example product line. The new product line generates twice as many products as the original one, but what matters is that half of them – the ones that do not have feature Copy – behave exactly as the original products. This ensures that the transformation is safe; we extended the product line without impacting existing users.

3.1 Formalization

We formalize these ideas in terms of product refinement (see Assumption 2). Basically, each program generated by the original product line must be refined by some program of the new, improved, product line.

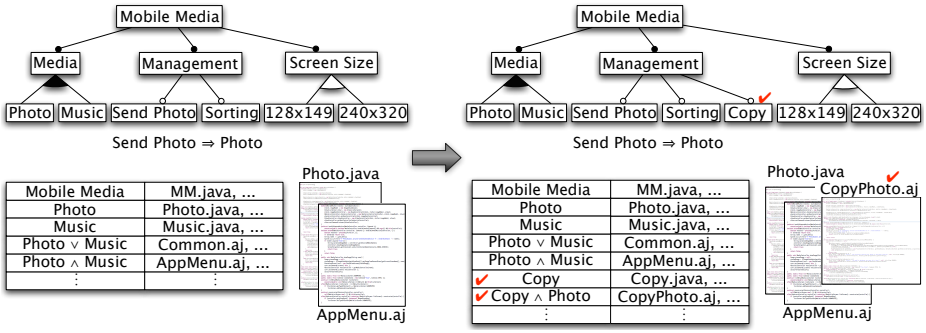

Fig. 4. Adding an optional feature refinement

Definition 7 ⟨Product line refinement⟩
For product lines (F, A, K) and (F', A', K'), the first is refined by the second, denoted

$$(F, A, K) \sqsubseteq (F', A', K')$$

whenever

$$\forall c \in [\![F]\!] \cdot \exists c' \in [\![F']\!] \cdot A\langle [\![K]\!] c\rangle \sqsubseteq A'\langle [\![K']\!] c'\rangle$$

Remember that, for a configuration c, a configuration knowledge K, and an asset mapping A related to a given product line, $A\langle [\![K]\!] c\rangle$ is a well-formed set of assets. So $A\langle [\![K]\!] c\rangle \sqsubseteq A'\langle [\![K']\!] c'\rangle$ refers to the product refinement notion discussed in Sec. 2.2.

3.2 Examples and considerations

To explore the definition just introduced, let us analyze a few concrete product line transformation scenarios.

Feature names do not matter First let us see how the definitions applies to the transformation depicted by Fig. 3. The feature models differ only by the name of a single feature. So they generate the same set of configurations, modulo renaming. For instance, for the source (left) product line configuration {Music, 240x320} we have the target (right) product line configuration {Audio, 240x320}. As the CKs have the same structure, evaluating them with these configurations yield

{`Commmon.aj`, `Music.java`, ...}

and

{`Commmon.aj`, `Audio.java`, ...}.

The resulting sets of asset names differ at most by a single element: `Audio.java` replacing `Music.java`. Finally, when applying these sets of names to both asset mappings, we obtain the same assets modulo global renaming, which is a well known refinement for closed programs. This is precisely what, by Definition 7, we need for assuring that the source product line is refined by the target product line.

This example shows that our refinement definition focus on the product line themselves, that is, the sets of products that can be generated. Contrasting with our previous notion of feature model refactoring [AGM+06], feature names do not matter. So users will not notice they are using products from the new product line, although developers might have to change their feature nomenclature when specifying product configurations. Not caring about feature names is essential for supporting useful refinements such as the just illustrated feature renaming and others that we discuss later.

Safety for existing users only To further explore the definitions, let us consider now the transformation shown in Fig. 4. The target feature model has an extra optional feature. So it generates all configurations of the source feature model plus extensions of these configurations with feature Copy. For example, it generates both {Music, 240x320} and {Music, 240x320, Copy}. For checking refinement, we focus only on the common configurations to both feature models – configurations without Copy. As the target CK is an extension of the source CK for dealing with cases when Copy is selected, evaluating the target CK with any configuration without Copy yields the same asset names yielded by the source CK with the same configuration. In this restricted name domain, both asset mappings are equal, since the target mapping is an extension of the first for names such as `CopyPhoto.java`, which appears only when Copy is selected. Therefore, the resulting assets produced by each product line are the same, trivially implying program refinement and then product line refinement.

By focusing on the common configurations to both feature models, we check nothing about the new products offered by the new product line. In fact, they might even not operate at all. Our refinement notion

assures only that users of existing products will not be disappointed by the corresponding products generated by the new product line. We give no guarantee to users of the new products, like the ones with Copy functionalities in our example. So refinements are safe transformations only in the sense that we can change a product line without impacting existing users.

Non refinements As discussed, the transformation depicted in Fig. 3 is a refinement. Classes and aspects are transformed by a global renaming, which preserves behavior for closed programs. But suppose that, besides renaming, we change the `AppMenu.aj`[2] aspect so that, instead of the menu on the left screenshot in Fig. 1, we have a menu with "Photos" and "Audio" options. The input-output behavior of new and original products would then not match, and users would observe the difference. So we would not be able to prove program refinement, nor product line refinement, consequently.

Despite not being a refinement, this menu change is an useful product line improvement, and should be carried on. The intention, however, is to change behavior, so developers will not be able to rely on the benefits of checking refinement. The benefits of checking for refinement only apply when the intention of the transformation is to improve product line configurability or internal structure, without changing observable behavior.

3.3 Basic properties

To support stepwise product line development and evolution, we now establish that product line refinement is a pre-order.

Theorem 9 ⟨Product line refinement reflexivity⟩

$$\forall l : ProductLine \cdot l \sqsubseteq l$$

Proof: Let $l = (F, A, K)$ be an arbitrary product line. By Definition 7, we have to prove that

$$\forall c \in [\![F]\!] \cdot \exists c' \in [\![F]\!] \cdot A\langle [\![K]\!]c\rangle \sqsubseteq A\langle [\![K]\!]c'\rangle$$

For an arbitrary $c \in [\![F]\!]$, just let c' be c and the proof follows from product refinement reflexivity (Axiom 3). □

[2] See Sec. 2.3 for understanding the role this aspect plays.

Theorem 10 ⟨Product line refinement transitivity⟩

$$\forall l_1, l_2, l_3 : ProductLine \cdot l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3$$

Proof: Let $l_1 = (F_1, A_1, K_1), l_2 = (F_2, A_2, K_2), l_3 = (F_3, A_3, K_3)$ be arbitrary product lines. Assume that $l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3$. By Definition 7, this amounts to

$$\forall c_1 \in [\![F_1]\!] \wedge \exists c_2 \in [\![F_2]\!] \cdot A_1\langle [\![K_1]\!] c_1\rangle \sqsubseteq A_2\langle [\![K_2]\!] c_2\rangle \tag{3}$$

and

$$\forall c_2 \in [\![F_2]\!] \cdot \exists c_3 \in [\![F_3]\!] \cdot A_2\langle [\![K_2]\!] c_2\rangle \sqsubseteq A_3\langle [\![K_3]\!] c_3\rangle \tag{4}$$

We then have to prove that

$$\forall c_1 \in [\![F_1]\!] \cdot \exists c_3 \in [\![F_3]\!] \cdot A_1\langle [\![K_1]\!] c_1\rangle \sqsubseteq A_3\langle [\![K_3]\!] c_3\rangle$$

For an arbitrary $c_1 \in [\![F_1]\!]$, we have to prove that

$$\exists c_3 \in [\![F_3]\!] \cdot A_1\langle [\![K_1]\!] c_1\rangle \sqsubseteq A_3\langle [\![K_3]\!] c_3\rangle \tag{5}$$

Properly instantiating c_1 in 3, we have

$$\exists c_2 \in [\![F_2]\!] \cdot A_1\langle [\![K_1]\!] c_1\rangle \sqsubseteq A_2\langle [\![K_2]\!] c_2\rangle$$

Let c_2' be such c_2. Properly instantiating c_2' in 4, we have

$$\exists c_3 \in [\![F_3]\!] \cdot A_2\langle [\![K_2]\!] c_2'\rangle \sqsubseteq A_3\langle [\![K_3]\!] c_3\rangle$$

Let c_3' be such c_3. Then we have

$$A_1\langle [\![K_1]\!] c_1\rangle \sqsubseteq A_2\langle [\![K_2]\!] c_2'\rangle \wedge A_2\langle [\![K_2]\!] c_2'\rangle \sqsubseteq A_3\langle [\![K_3]\!] c_3'\rangle$$

By product refinement transitivity (Axiom 4), we have

$$A_1\langle [\![K_1]\!] c_1\rangle \sqsubseteq A_3\langle [\![K_3]\!] c_3'\rangle$$

This gives us the c_3 in 5 that completes our proof.

□

4 Product line refinement compositionality

The product line refinement notion allows one to reason about a product line as a whole, considering its three elements (artifacts): feature model, CK, and asset mapping. However, for independent development of product line artifacts, we must support separate and compositional reasoning for each product line artifact. This allows us to evolve product line artifacts independently. We first consider feature models. Replacing a feature model by an equivalent one leads to a refined product line.

Theorem 11 ⟨Feature model equivalence compositionality⟩
For product lines (F, A, K) and (F', A, K), if

$$F \cong F'$$

then

$$(F, A, K) \sqsubseteq (F', A, K)$$

Proof: For arbitrary F, F', A, K, assume that $F \cong F'$. By Definition 7, we have to prove that

$$\forall c \in [\![F]\!] \cdot \exists c' \in [\![F']\!] \cdot A\langle [\![K]\!] c\rangle \sqsubseteq A\langle [\![K]\!] c'\rangle$$

From our assumption and Definition 1, this is equivalent to

$$\forall c \in [\![F]\!] \cdot \exists c' \in [\![F]\!] \cdot A\langle [\![K]\!] c\rangle \sqsubseteq A\langle [\![K]\!] c'\rangle$$

For an arbitrary $c \in [\![F]\!]$, just let c' be c and the proof follows from product refinement reflexivity (Axiom 4). □

We require feature model equivalence because feature model refinement, which requires $[\![F]\!] \subseteq [\![F']\!]$ instead of $[\![F]\!] = [\![F']\!]$, is not enough for ensuring that separate modifications to a feature model imply refinement for the product line. In fact, refinement allows the new feature model to have extra configurations that might not generate valid products; the associated feature model refinement transformation would not lead to a valid product line. For example, consider that the extra configurations result from eliminating an alternative constraint between two features, so that they become optional. The assets that implement these features might well be incompatible, generating an invalid program when both features are selected. Refinement of the whole product line, in this case, would also demand changes to the assets and CK.

We can also independently evolve a CK. For similar reasons, we require CK equivalence as well.

Theorem 12 ⟨CK equivalence compositionality⟩
For product lines (F, A, K) and (F, A, K'), if

$$K \cong K'$$

then

$$(F, A, K) \sqsubseteq (F, A, K')$$

Proof: The proof is similar to that of Theorem 11, using Definition 2 instead of Definition 1. □

Note that the reverse does not hold because the asset names generated by K and K' might differ for assets that have no impact on product behavior,[3] or for assets that have equivalent behavior but are named differently in the product lines. For similar reasons, the reverse does not hold for Theorem 11.

For asset mappings, we can rely only on refinement. Separately refining an asset mapping implies refinement for the product line as a whole.

Theorem 13 ⟨Asset mapping refinement compositionality⟩
For product lines (F, A, K) and (F, A', K), if

$$A \sqsubseteq A'$$

then

$$(F, A, K) \sqsubseteq (F, A', K)$$

Proof: For arbitrary F, A, A', and K, assume that $A \sqsubseteq A'$. By Definition 7, we have to prove that

$$\forall c \in [\![F]\!] \cdot \exists c' \in [\![F]\!] \cdot A\langle [\![K]\!]c\rangle \sqsubseteq A'\langle [\![K]\!]c'\rangle$$

For an arbitrary $c \in [\![F]\!]$, if we prove

$$A\langle [\![K]\!]c\rangle \sqsubseteq A'\langle [\![K]\!]c\rangle \tag{6}$$

then c is the necessary c' we need to complete the proof. By Lemma 5 and our assumption, we have that

$$\begin{aligned} &\forall ans : fset[AssetName] \cdot \forall as : fset[Asset] \cdot \\ &\qquad wf(as \cup A\langle ans\rangle) \\ &\quad \Rightarrow wf(as \cup A'\langle ans\rangle) \wedge as \cup A\langle ans\rangle \sqsubseteq as \cup A'\langle ans\rangle \end{aligned} \tag{7}$$

[3] Obviously an anomaly, but still possible.

By properly instantiating ans with $[\![K]\!]c$ and as with $\{\}$ in 7, from set union properties we obtain

$$\begin{array}{l} \quad wf(A\langle[\![K]\!]c\rangle) \\ \Rightarrow wf(A'\langle[\![K]\!]c\rangle) \wedge A\langle[\![K]\!]c\rangle \sqsubseteq A'\langle[\![K]\!]c\rangle \end{array} \tag{8}$$

From Definition 6, we have that $wf(A\langle[\![K]\!]c\rangle)$ for all $c \in [\![F]\!]$. Therefore, from this and 8 we obtain

$$wf(A'\langle[\![K]\!]c\rangle) \wedge A\langle[\![K]\!]c\rangle \sqsubseteq A'\langle[\![K]\!]c\rangle$$

concluding the proof (see 6). □

Finally, we have the full compositionality theorem, which justifies completely independent development of product line artifacts.

Theorem 14 ⟨Full compositionality⟩
For product lines (F, A, K) and (F', A', K'), if

$$F \cong F' \wedge A \sqsubseteq A' \wedge K \cong K'$$

then

$$(F, A, K) \sqsubseteq (F', A', K')$$

Proof: First assume that $F \cong F'$, $A \sqsubseteq A'$, and $K \cong K'$. By Lemma 6, the fact that (F, A, K) is a product line, and Definition 6, we have that (F', A, K) is a product line. Then, using Theorem 11, we have

$$(F, A, K) \sqsubseteq (F', A, K) \tag{9}$$

Similarly, from our assumptions, deductions, and Lemma 7 we have that (F', A, K') is a product line. Using Theorem 12, we have

$$(F', A, K) \sqsubseteq (F', A, K') \tag{10}$$

Again, from our assumptions, deductions, and Lemma 8, we have that (F', A', K') is a product line. Using Theorem 13, we have

$$(F', A, K') \sqsubseteq (F', A', K') \tag{11}$$

The proof then follows from 9, 10, 11, and product line refinement transitivity (Theorem 10). □

5 Related work

The notion of product line refinement discussed here first appeared in a product line refactoring tutorial [Bor09]. Besides talking about product line and population refactoring, this tutorial illustrates different kinds of refactoring transformation templates that can be useful for deriving and evolving product lines. In this paper we extend the initial formalization of the tutorial making clear the interface between our theory and languages used to describe product line artifacts. We also derive a number of properties that were not explored in the tutorial. We encode the theory in the PVS specification language and prove all properties with the PVS prover.

Our notion of product line refinement goes beyond refactoring of feature models [AGM+06,GMB08], considering also other artifacts like configuration knowledge and assets, both in isolation and in an integrated way. In particular, the refinement notion explored here is independent of the language used to describe feature models. The cited formalization of feature models [AGM+06,GMB08], and others [SHTB07], could, however, be used to instantiate our theory for dealing with specific feature model notation and semantics. Similarly, our theory is independent of product refinement notions. A program refinement notion, like the one for a sequential subset of Java [SB04,BSCC04], could be used to instantiate our general theory.

Early work [CDCvdH03] on product line refactoring focus on Product Line Architectures (PLAs) described in terms of high-level components and connectors. This work presents metrics for diagnosing structural problems in a PLA, and introduces a set of architectural refactorings that can be used to resolve these problems. Besides being specific to architectural assets, this work does not deal with other product line artifacts such as feature models and configuration knowledge. There is also no notion of behavior preservation for product lines, as captured here by our notion of product line refinement.

Several approaches [KMPY05,TBD06,LBL06,KAB07] focus on refactoring a product into a product line, not exploring product line evolution in general, as we do here. First, Kolb et al. [KMPY05] discuss a case study in refactoring legacy code components into a product line implementation. They define a systematic process for refactoring products with the aim of obtaining product lines assets. There is no discussion about feature models and configuration knowledge. Moreover, behavior preservation and configurability of the resulting product lines are only checked by testing. Similarly, Kastner et al. [KAB07] focus only on transforming

code assets, implicitly relying on refinement notions for aspect-oriented programs [CB05]. As discussed here and elsewhere [Bor09] these are not adequate for justifying product line refinement and refactoring. Trujillo et al. [TBD06] go beyond code assets, but do not explicitly consider transformations to feature model and configuration knowledge. They also do not consider behavior preservation; they indeed use the term "refinement", but in the quite different sense of overriding or adding extra behavior to assets.

Liu et al. [LBL06] 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 base asset (a code module, for instance) and related derivative assets, which contain feature declarations appropriate for different product configurations. Contrasting with our theory, this theory does not consider feature model transformations and assumes an implicit notion of configuration knowledge based on the idea of derivatives. So it does not consider explicit configuration knowledge transformations as we do here. Their work is, however, complementary to ours since we abstract from specific asset transformation techniques such as the one supported by their theory. By proving that their technique can be mapped to our notion of asset refinement, both theories could be used together.

The theory we present in this paper aims to formalize concepts and processes from tools [LBL06,CBS+07,ACN+08] and practical experience [ACV+05,AJC+05,KMPY05,AGM+06,TBD06,KAB07] on product line refactoring. A more rigorous evaluation of the proposed theory is, however, left as future work.

6 Conclusions

In this paper we present a general theory of product line refinement, formalizing refinement and equivalence notions for product lines and its artifacts: feature model, configuration knowledge, and asset mapping. More important, we establish a number of properties that justify stepwise and compositional product line development and evolution. The presented theory is largely independent of the languages used to describe feature model, configuration knowledge, and reusable assets. We make this explicit through assumptions and axioms about basic concepts related to these languages.

By instantiating this theory with proper notations and semantic formalizations for feature models and the other product line artifacts, we can directly use the refinement and equivalence notions, and the associated properties, to guide and improve safety of the product line derivation and evolution processes. Such an instantiation also allows one to formally prove soundness of product line refactoring transformation templates [Bor09] expressed in those notations. As the transformation templates precisely specify the transformation mechanics and preconditions, their soundness is specially useful for correctly implementing the transformations and avoiding typical problems with current program refactoring tools [ST09]. In fact, soundness could help to avoid even subtler problems that can appear with product line refactoring tools.

Acknowledgements

We would like to thank colleagues of the Software Productivity Group for helping to significantly improve this work. Márcio Ribeiro provided support with the Mobile Media example. Rodrigo Bonifácio played a fundamental role developing the configuration knowledge approach used here. Vander Alves and Tiago Massoni carried on the initial ideas about feature model and product line refactoring. Fernando Castor, Carlos Pontual, Sérgio Soares, Rodrigo, and Márcio provided excellent feedback on early versions of part of the material presented here. We would also like to acknowledge current financial support from CNPq, FACEPE, and CAPES projects, and early support from FINEP and Meantime mobile creations.

A Extra proofs

In this appendix we present the proofs we omitted in the main text. The PVS specification of the whole theory, and proof files for all lemmas and theorems are available at `http://twiki.cin.ufpe.br/twiki/bin/view/SPG/TheorySPLRefinement`.

Lemma 1 ⟨Distributed mapping over union⟩
For asset mapping A, asset a, and finite sets of asset names S and S', if

$$a \in A\langle S \cup S' \rangle$$

then

$$a \in A\langle S \rangle \vee a \in A\langle S' \rangle$$

Proof: For arbitrary A, a, S , and S', assume $a \in A\langle S \cup S'\rangle$. From this and Definition 4 ($A\langle\rangle$) we have

$$a \in \{a : Asset \mid \exists n \in S \cup S' \cdot (n, a) \in m\}$$

From set union and membership properties, we have

$$a \in \{a : Asset \mid \exists n \in S \cdot (n, a) \in m \vee \exists n \in S' \cdot (n, a) \in m\}$$

From set comprehension properties, we have

$$a \in \{a : Asset \mid \exists n \in S \cdot (n, a) \in m\} \cup \{a : Asset \mid \exists n \in S' \cdot (n, a) \in m\}$$

By applying twice Definition 4 ($A\langle\rangle$), we derive

$$a \in A\langle S\rangle \cup A\langle S'\rangle$$

The proof follows from the above and set membership properties. □

Lemma 2 ⟨Distributed mapping over singleton⟩
For asset mapping A, asset name an, and finite set of asset names S, if

$$an \in dom(A)$$

then

$$\exists a : Asset \cdot (an, a) \in A \ \wedge \ A\langle an \cup S\rangle = a \cup A\langle S\rangle$$

Proof: For arbitrary A, an, and S, assume $an \in dom(A)$. From this, Definition 4 (dom), and set comprehension and membership properties, we have

$$\exists a : Asset \cdot (an, a) \in A \tag{12}$$

Let a_1 be such a. By Definition 4 ($A\langle\rangle$), we have

$$A\langle an \cup S\rangle = \{a : Asset \mid \exists n \in an \cup S \cdot (n, a) \in A\}$$

Again, by set membership and comprehension properties, we have

$$\begin{aligned} A\langle an \cup S\rangle = \\ &\{a : Asset \mid \exists n \in \{an\} \cdot (n, a) \in A\} \\ \cup &\{a : Asset \mid \exists n \in S \cdot (n, a) \in A\} \end{aligned}$$

By Definition 4 ($A\langle\rangle$), our assumption that A is an asset mapping, and set membership and comprehension properties, we have

$$A\langle\{an\} \cup S\rangle = a_1 \cup A\langle S\rangle$$

From this and remembering that 12 was instantiated with a_1, a_1 provides the a we need to conclude the proof.

□

Lemma 3 ⟨Asset mapping domain membership⟩
For asset mapping A, asset name an, and asset a, if

$$(an, a) \in A$$

then

$$an \in dom(A)$$

Proof: For arbitrary A, an, and a, assume $(an, a) \in A$. By Definition 4 (dom), we have to prove that

$$\exists x : Asset \mid (an, x) \in A$$

Let x be a, and this concludes the proof. □

Lemma 4 ⟨Distributed mapping over set of non domain elements⟩
For asset mapping A and finite set of asset names S, if

$$\neg \exists n \in S \cdot n \in dom(A)$$

then

$$A\langle S \rangle = \{\}$$

Proof: For arbitrary A and S, assume $\neg \exists n \in S \cdot n \in dom(A)$. By Definition 4 ($A\langle\rangle$), we have to prove that

$$\{a : Asset \mid \exists n \in S \cdot (n, a) \in A\} = \{\}$$

By Lemma 3, we then have to prove that

$$\{a : Asset \mid \exists n \in S \cdot n \in dom(A) \wedge (n, a) \in A\} = \{\}$$

The proof follows from the above, our assumption, and set comprehension properties. □

Lemma 5 ⟨Asset mapping compositionality⟩
For asset mapping A and A', if

$$A \sqsubseteq A'$$

then

$$\begin{aligned}&\forall ans : fset[AssetName] \cdot \forall as : fset[Asset] \cdot \\ &\quad wf(as \cup A\langle ans\rangle) \\ &\Rightarrow wf(as \cup A'\langle ans\rangle) \wedge as \cup A\langle ans\rangle \sqsubseteq as \cup A'\langle ans\rangle\end{aligned}$$

Proof: For arbitrary A and A', assume $A \sqsubseteq A'$. From Definition 5, we have

$$\begin{aligned}&dom(A) = dom(A') \\ &\wedge \forall n \in dom(A) \cdot \\ &\quad \exists a, a' : Asset \cdot (n, a) \in A \wedge (n, a') \in A' \wedge a \sqsubseteq a'\end{aligned} \tag{13}$$

By induction on the cardinality of ans, assume the induction hypothesis

$$\begin{aligned}&\forall ans' : fset[AssetName] \cdot \\ &\quad card(ans') < card(ans) \\ &\Rightarrow \forall as : fset[Asset] \cdot \\ &\quad wf(as \cup A\langle ans'\rangle) \\ &\Rightarrow wf(as \cup A'\langle ans'\rangle) \wedge as \cup A\langle ans'\rangle \sqsubseteq as \cup A'\langle ans\rangle\end{aligned} \tag{14}$$

and we have to prove

$$\begin{aligned}&\forall as : fset[Asset] \cdot \\ &\quad wf(as \cup A\langle ans\rangle) \\ &\Rightarrow wf(as \cup A'\langle ans\rangle) \wedge as \cup A\langle ans\rangle \sqsubseteq as \cup A'\langle ans\rangle\end{aligned} \tag{15}$$

By case analysis, now consider that $\neg(\exists an \in ans \cdot an \in dom(A))$. By Lemma 4, we have that $A\langle ans\rangle = \{\}$. Similarly, given that $dom(A) = dom(A')$ (see 13), we also have that $A'\langle ans\rangle = \{\}$. So, by set union properties, we are left to prove that

$$\forall as : fset[Asset] \cdot wf(as) \Rightarrow wf(as) \wedge as \sqsubseteq as$$

The proof trivially follows from Axiom 3 and propositional calculus.

Let's now consider the case $\exists an \in ans \cdot an \in dom(A)$. By basic set properties, we have that $ans = an \cup ans'$ for some asset name $an \in dom(A)$ and set ans' such that $an \notin ans'$. Then, from 15, we are left to prove that

$$\begin{aligned}&\forall as : fset[Asset] \cdot \\ &\quad wf(as \cup A\langle an \cup ans'\rangle) \\ &\Rightarrow \quad wf(as \cup A'\langle an \cup ans'\rangle) \\ &\quad \wedge as \cup A\langle an \cup ans'\rangle \sqsubseteq as \cup A'\langle an \cup ans'\rangle\end{aligned}$$

By Lemma 2, given that $an \in dom(A)$ and consequently $an \in dom(A')$, we have that $A\langle an \cup ans' \rangle = a \cup A\langle ans' \rangle$ and $A'\langle an \cup ans' \rangle = a' \cup A'\langle ans' \rangle$ for some assets a and a'. From 13, we also have that $a \sqsubseteq a'$. By equational reasoning, we then have to prove that

$$\begin{array}{l} \forall as : fset[Asset] \cdot \\ \quad\quad wf(as \cup a \cup A\langle ans' \rangle) \\ \quad \Rightarrow \quad wf(as \cup a' \cup A'\langle ans' \rangle) \\ \quad\quad \wedge\, as \cup a \cup A\langle ans' \rangle \sqsubseteq as \cup a' \cup A'\langle ans' \rangle \end{array}$$

For an arbitrary as, assume $wf(as \cup a \cup A\langle ans' \rangle)$ and then we have to prove that

$$\begin{array}{l} \quad wf(as \cup a' \cup A'\langle ans' \rangle) \\ \wedge\, as \cup a \cup A\langle ans' \rangle \sqsubseteq as \cup a' \cup A'\langle ans' \rangle \end{array} \tag{16}$$

By the induction hypothesis (see 14), instantiating ans' with the ans' just introduced, note that we will have $card(ans') < card(ans)$ and, therefore

$$\begin{array}{l} \forall as : fset[Asset] \cdot \\ \quad\quad wf(as \cup A\langle ans' \rangle) \\ \quad\quad \Rightarrow wf(as \cup A'\langle ans' \rangle) \wedge as \cup A\langle ans' \rangle \sqsubseteq as \cup A'\langle ans \rangle \end{array}$$

From this, instantiating as as $as \cup a$, and remembering that we have already assumed $wf(as \cup a \cup A\langle ans' \rangle)$, we have

$$\begin{array}{l} \quad wf(as \cup A'\langle ans' \rangle) \\ \wedge\, as \cup A\langle ans' \rangle \sqsubseteq as \cup A'\langle ans \rangle \end{array}$$

Now, given that $a \sqsubseteq a'$, from the compositionality axiom (Axiom 5) and the above we have that

$$\begin{array}{l} \quad wf(()as \cup a' \cup A'\langle ans' \rangle) \\ \wedge\, as \cup a \cup A'\langle ans' \rangle \sqsubseteq as \cup a' \cup A'\langle ans' \rangle \end{array}$$

The proof then follows from 16, the above, and Axiom 4. □

Lemma 6 ⟨Feature model equivalence compositionality over wf⟩
For feature models F and F', asset mapping A, and configuration knowledge K, if

$$F \cong F' \wedge \forall c \in [\![F]\!] \cdot wf(A\langle [\![K]\!] c \rangle)$$

then

$$\forall c \in [\![F']\!] \cdot wf(A\langle [\![K]\!]c\rangle)$$

Proof: For arbitrary F, F', A, and K, assume

$$F \cong F' \wedge \forall c \in [\![F]\!] \cdot wf(A\langle [\![K]\!]c\rangle)$$

By Definition 1, what we have to prove is equivalent to

$$\forall c \in [\![F]\!] \cdot wf(A\langle [\![K]\!]c\rangle)$$

which corresponds to our assumption. □

Lemma 7 ⟨CK equivalence compositionality over wf⟩
For feature model F, asset mapping A, and configuration knowledge K and K', if

$$K \cong K' \wedge \forall c \in [\![F]\!] \cdot wf(A\langle [\![K]\!]c\rangle)$$

then

$$\forall c \in [\![F]\!] \cdot wf(A\langle [\![K']\!]c\rangle)$$

Proof: Similar to proof of Lemma 6, using Definition 2 instead. □

Lemma 8 ⟨Asset mapping refinement compositionality over wf⟩
For feature model F, asset mappings A and A', and configuration knowledge K, if

$$A \sqsubseteq A' \wedge \forall c \in [\![F]\!] \cdot wf(A\langle [\![K]\!]c\rangle)$$

then

$$\forall c \in [\![F]\!] \cdot wf(A'\langle [\![K]\!]c\rangle)$$

Proof: For arbitrary F, A, A', and K, assume

$$A \sqsubseteq A' \wedge \forall c \in [\![F]\!] \cdot wf(A\langle [\![K]\!]c\rangle) \tag{17}$$

For an arbitrary $c \in [\![F]\!]$, we then have to prove that

$$wf(A'\langle [\![K]\!]c\rangle) \tag{18}$$

By properly instantiating the assumption (17) with the just introduced c, we have

$$wf(A\langle[\![K]\!]c\rangle) \tag{19}$$

From Lemma 5 and the assumption (17), we have

$$\begin{aligned}
&\forall ans : fset[AssetName] \cdot \forall as : fset[Asset] \cdot \\
&\quad wf(as \cup A\langle ans\rangle) \\
&\Rightarrow wf(as \cup A'\langle ans\rangle) \wedge \\
&\quad as \cup A\langle ans\rangle \sqsubseteq as \cup A'\langle ans\rangle
\end{aligned}$$

Instantiating ans with $[\![K]\!]c$, as with $\{\}$, and by set union properties, we have

$$\begin{aligned}
&wf(A\langle[\![K]\!]c\rangle) \\
&\Rightarrow wf(A'\langle[\![K]\!]c\rangle) \wedge A\langle[\![K]\!]c\rangle \sqsubseteq A'\langle[\![K]\!]c\rangle
\end{aligned}$$

The proof (see 18) then follows from the above and 19. □

References

ACN+08. Vander Alves, Fernando Calheiros, Vilmar Nepomuceno, Andrea Menezes, Sérgio Soares, and Paulo Borba. FLiP: Managing software product line extraction and reaction with aspects. In *12th International Software Product Line Conference*, page 354. IEEE Computer Society, 2008.

ACV+05. Vander Alves, Ivan Cardim, Heitor Vital, Pedro Sampaio, Alexandre Damasceno, Paulo Borba, and Geber Ramalho. Comparative analysis of porting strategies in J2ME games. In *21st IEEE International Conference on Software Maintenance*, pages 123–132. IEEE Computer Society, 2005.

AGM+06. Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos Lucena. Refactoring product lines. In *5th International Conference on Generative Programming and Component Engineering*, pages 201–210. ACM, 2006.

AJC+05. Vander Alves, Pedro Matos Jr., Leonardo Cole, Paulo Borba, and Geber Ramalho. Extracting and evolving mobile games product lines. In *9th International Software Product Line Conference*, volume 3714 of *LNCS*, pages 70–81. Springer-Verlag, 2005.

Bat05. Don Batory. Feature models, grammars, and propositional formulas. In *9th International Software Product Lines Conference*, volume 3714 of *LNCS*, pages 7–20. Springer-Verlag, 2005.

BB09. Rodrigo Bonifácio and Paulo Borba. Modeling scenario variability as crosscutting mechanisms. In *8th International Conference on Aspect-Oriented Software Development*, pages 125–136. ACM, 2009.

Bor09. Paulo Borba. An introduction to software product line refactoring. In *3rd Summer School on Generative and Transformational Techniques in Software Engineering (to appear)*, 2009.

BSCC04. Paulo Borba, Augusto Sampaio, Ana Cavalcanti, and Márcio Cornélio. Algebraic reasoning for object-oriented programming. *Science of Computer Programming*, 52:53–100, 2004.

CB05. Leonardo Cole and Paulo Borba. Deriving refactorings for AspectJ. In *4th International Conference on Aspect-Oriented Software Development*, pages 123–134. ACM, 2005.

CBS+07. Fernando Calheiros, Paulo Borba, Sérgio Soares, Vilmar Nepomuceno, and Vander Alves. Product line variability refactoring tool. In *1st Workshop on Refactoring Tools*, pages 33–34, July 2007.

CDCvdH03. Matt Critchlow, Kevin Dodd, Jessica Chou, and André van der Hoek. Refactoring product line architectures. In *1st International Workshop on Refactoring: Achievements, Challenges, and Effects*, pages 23–26, 2003.

CE00. Krysztof Czarnecki and Ulrich Eisenecker. *Generative programming: methods, tools, and applications*. Addison-Wesley, 2000.

CHE05. Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

CN01. Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

FCS+08. Eduardo Figueiredo, Nélio Cacho, Claudio Sant'Anna, Mário Monteiro, Uirá Kulesza, Alessandro Garcia, Sérgio Soares, Fabiano Ferrari, Safoora Khan, Fernando Filho, and Francisco Dantas. Evolving software product lines with aspects: an empirical study on design stability. In *30th International Conference on Software Engineering*, pages 261–270. ACM, 2008.

Fow99. Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

GA01. Cristina Gacek and Michalis Anastasopoulos. Implementing product line variabilities. *SIGSOFT Software Engineering Notes*, 26(3):109–117, 2001.

GMB05. Rohit Gheyi, Tiago Massoni, and Paulo Borba. An abstract equivalence notion for object models. *Electronic Notes in Theoretical Computer Science*, 130:3–21, 2005.

GMB08. Rohit Gheyi, Tiago Massoni, and Paulo Borba. Algebraic laws for feature models. *Journal of Universal Computer Science*, 14(21):3573–3591, 2008.

KAB07. Christian Kastner, Sven Apel, and Don Batory. A case study implementing features using AspectJ. In *11th International Software Product Line Conference*, pages 223–232. IEEE Computer Society, 2007.

KCH+90. Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

KMPY05. Ronny Kolb, Dirk Muthig, Thomas Patzke, and Kazuyuki Yamauchi. A case study in refactoring a legacy component for reuse in a product line. In *21st International Conference on Software Maintenance*, pages 369–378. IEEE Computer Society, 2005.

Kru02. Charles Krueger. Easing the transition to software mass customization. In *4th International Workshop on Software Product-Family Engineering*, volume 2290 of *LNCS*, pages 282–293. Springer-Verlag, 2002.

LBL06. Jia Liu, Don Batory, and Christian Lengauer. Feature oriented refactoring of legacy applications. In *28th International Conference on Software Engineering*, pages 112–121. ACM, 2006.

MGB08. Tiago Massoni, Rohit Gheyi, and Paulo Borba. Formal model-driven program refactoring. In *11th International Conference on Fundamental Approaches to Software Engineering*, volume 4961 of *LNCS*, pages 362–376. Springer-Verlag, 2008.

Opd92. William Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

ORS92. Sam Owre, John Rushby, and Natarajan Shankar. Pvs: A prototype verification system. In *11th International Conference on Automated Deduction*, pages 748–752. Springer-Verlag, 1992.

PBvdL05. Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

SB04. Augusto Sampaio and Paulo Borba. Transformation laws for sequential object-oriented programming. In *Refinement Techniques in Software Engineering*, volume 3167 of *LNCS*, pages 18–63. Springer, 2004.

SHTB07. Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, 2007.

ST09. Friedrich Steimann and Andreas Thies. From public to private to absent: Refactoring Java programs under constrained accessibility. In *23rd European Conference on Object-Oriented Programming*, volume 5653 of *LNCS*, pages 419–443. Springer, 2009.

TBD06. Salvador Trujillo, Don Batory, and Oscar Diaz. Feature refactoring a multi-representation program into a product line. In *5th International Conference on Generative Programming and Component Engineering*, pages 191–200. ACM, 2006.

vdLSR07. Frank van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: the Best Industrial Practice in Product Line Engineering*. Springer, 2007.