Theoretical Computer Science 455 (2012) 2–30

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

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

A theory of software product line refinement

Paulo Borba [a,*], Leopoldo Teixeira [a], Rohit Gheyi [b]

[a] *Informatics Center, Federal University of Pernambuco, Recife, PE, Brazil*
[b] *Department of Computing Systems, Federal University of Campina Grande, Campina Grande, PB, Brazil*

ARTICLE INFO

Keywords:
Software product lines
Software evolution
Refinement
Refactoring

ABSTRACT

To safely evolve a software product line, it is important to have a notion of product line refinement that assures behavior preservation of the original product line products. So in this article we present a language independent theory of product line refinement, establishing refinement properties that justify stepwise and compositional product line evolution. Moreover, we instantiate our theory with the formalization of specific languages for typical product lines artifacts, and then introduce and prove soundness of a number of associated product line refinement transformation templates. These templates can be used to reason about specific product lines and as a basis to derive comprehensive product line refinement catalogues.

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 we compose or instantiate 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 [1,2].

To obtain these benefits with reduced upfront investment, previous work [3–5] proposes to minimize the initial product line analysis and development process by bootstrapping existing related products into a product line. In this context, it is important to rely on notions of product line and product population refactoring [6], which provide guidance and safety for deriving a product line from existing products. These notions are also important for safely evolving a product line by simply improving its design or even by adding new products while preserving existing ones. To support these activities, product line refactoring goes beyond program refactoring notions [7–10], which consider behavior preservation and quality improvement of single programs only. For product lines, refactoring considers extra artifacts, such as feature models [11,12], which are used for automatically generating products from assets. Moreover, we must deal with sets of reusable assets that do not necessarily correspond to valid programs; in fact, a product line might even have conflicting assets due to alternative features and different kinds of feature interaction.

Instead of dealing directly with the stronger notion of refactoring, we focus on the underlying notion of refinement [13,14], which also captures behavior preservation but abstracts quality improvement. We take the broader view of refinement as a relation that preserves properties necessary to assure safe evolution. So we demand only behavior preservation of the original product line elements, allowing the product line to be extended with new products as long as our demand is satisfied. The focus on refinement allows us to develop a formal theory of product line refinement [15], which is general with respect to the different languages that we can use to describe or implement a product line. To improve generality, in this article we slightly revise that theory. More importantly, to illustrate one of the main

[*] Corresponding author.
E-mail addresses: phmb@cin.ufpe.br (P. Borba), lmt@cin.ufpe.br (L. Teixeira), rohit@dsc.ufcg.edu.br (R. Gheyi).

applications of our refinement theory, we extend our previous work in two ways. First, we instantiate the revised general theory with formalizations of specific languages for feature models and configuration knowledge [12], which are used to automatically generate products from assets. Second, using these languages, we formalize a number of product line refinement transformation templates and prove their soundness with respect to our theory's refinement notion. We encode our theory and the mentioned languages in the Prototype Verification System (PVS) [16], which we use to prove soundness.

With the soundness result, we can assure that the refinement transformation templates precisely capture different kinds of changes that can be safely performed when evolving a product line described with the languages we formalize here. They go beyond templates for transforming feature models [17,18], considering also configuration knowledge and assets, in an integrated way. In this way, we provide support to reason about specific product lines, and establish the basis for the derivation of a comprehensive product line refinement catalogue, specific to the languages adopted here. By providing a concrete instantiation of the general theory, this work might be also useful to others interested in deriving product line transformation templates considering other notations and semantic formalizations for product line artifacts [19–21,18].

This text is organized as follows. Section 2 introduces and formalizes basic concepts and notation for feature models, assets and configuration knowledge [12,22]. Following that, in Section 3, we introduce our product line refinement theory, including our notion of product line refinement and properties that justify stepwise and compositional product line evolution. Assumptions and axioms explicitly establish the interfaces between our theory and particular languages used to describe a product line. Section 4 shows how we instantiate our theory with the language formalizations of Section 2, and then presents a number of refinement templates and the associated soundness results.[1] We discuss related work in Section 5 and conclude with Section 6.

2. Product lines concepts

To enable the automatic generation of products from assets, product lines (PLs) often rely on artifacts such as Feature Models (FMs) and Configuration Knowledge (CK) [12]. A FM specifies common and variant features among products, so we can use it to describe and select products based on the features they support. A simple and explicit CK representation, for example, relates features and assets, specifying which assets specify or implement possible feature combinations. Hence we use a CK to actually build a product given chosen features for that product.

As better explained later, our PL refinement theory can also consider alternatives to FM and CK. Roughly, as long as we have ways to express the set of valid configurations of a PL, and how products are built given a valid configuration, we can apply our theory. However, for making the discussion more concrete, we use basic notations and terminology for FM and CK when explaining the theory concepts. So we now explain in more detail these two kinds of artifacts and other concepts, using examples from the Mobile Media PL [23], which contains applications – such as the one illustrated in Fig. 1 – that manipulate photos, music, and video on mobile devices.

2.1. Feature models

A FM is usually represented as a tree, containing features and information about how they relate to each other. Features have different names and basically abstract groups of associated requirements, both functional and non-functional. In the FM 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 FM, where Sorting is optional, Media is mandatory, Photo and Music are or-features, and the two illustrated screen sizes are alternative.

Besides these relationships, the FM notation we consider here may also contain propositional logic formulae about features. We use feature names as atoms to indicate feature selection. 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 we select feature Send Photo. 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}

[1] The online appendix containing the proofs not detailed in the text and the complete theory and proofs in PVS is available at: http://twiki.cin.ufpe.br/twiki/bin/view/SPG/TheorySPLRefinement.

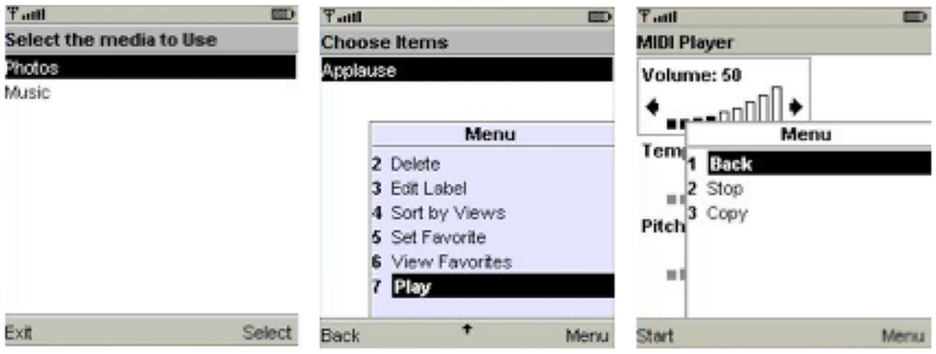


Fig. 1. Mobile Media screenshots.

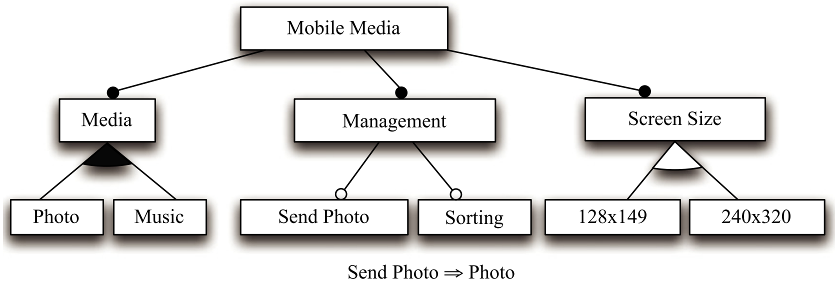


Fig. 2. Mobile Media simplified feature model

is not because it breaks the Screen Size alternative constraint. In summary, a valid configuration is one that satisfies all FM constraints, specified both graphically and through formulae. Each valid configuration corresponds to one PL product, expressed in terms of the features it supports. This captures the intuition that a FM denotes the set of products in a PL.

To simplify formalization of the FM notation we use here, we translate graphical constraints in a FM to logical propositions [20,18]. Each kind of relationship translates to a particular kind of proposition. For example, we represent a mandatory relationship between a child M of a feature F as $M \Leftrightarrow F$, as in Media $\Leftrightarrow$ Mobile Media. As another example, we translate the optional relationship between a child O of a feature F as $O \Rightarrow F$. The translation defined by transformations like that [18] allows us to formalize FMs as a set of feature names and a set of propositional formulae, which includes both the relationship constraints and the additional formulae over features. We express FMs and other useful types and predicates in the following, using a simplified PVS notation where, for example, $< \ldots >$ represents record types and $\mathcal{P}$ represents the powerset operation:

Definition 1 (*Feature Models*).

$$
\begin{aligned}
&Name : TYPE\\
&FM : < features : \mathcal{P}[Name], formulae : \mathcal{P}[Formula] >\\
&Configuration : \mathcal{P}[Name]\\
&wfFM(fm : FM) : bool = \forall f \in formulae(fm) \cdot wt(f, fm)\\
&WFM : TYPE = (wfFM)\\
&semantics(fm : WFM) : \mathcal{P}[Configuration] =\\
&\quad \{c \mid c \subseteq features(fm) \wedge \forall f \in formulae(fm) \cdot satisfies(f, c)\}.
\end{aligned}
$$

We use *TYPE* to declare an uninterpreted type that imposes no assumptions on instantiations of this FM theory. We define FMs using record types. Contrasting to *TYPE*, record types are empty if any of its component types is empty, since the resulting type corresponds to the cartesian products of their constituents. By using record types, we can access the feature set of a feature model *fm* using *features*(*fm*). We do likewise for the formulae. The FM formula language contains six kinds of propositional formulae: true, false, feature name, negation, conjunction, and implication. We represent these with PVS abstract datatypes [18], which we omit here for brevity. A configuration is a set of names, representing the selected features. The $wfFM$ function checks whether all formulae are well-typed. The wt function asserts that a formula only contain names from the given FM. We define *WFM* as the subtype of FMs that are well-formed.

Finally, we can obtain the set of all valid configurations from a FM, which is often defined as the semantics of a FM. We define the *satisfies* function as a recursive function that evaluates a propositional formula against a configuration. For instance, a configuration satisfies the conjunction formula $p \wedge q$ if it satisfies p and q. 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 of this encoding and the others discussed in this article, but they are available in the online appendix.

We could have used here alternative FM notations and semantic encodings [19–21,18], but, as shall be clear later, our theory of PL refinement does not depend on the FM notation we use. We can instantiate our theory with any FM notation that expresses its semantics as a set of configurations. For example, this is the case of cardinality-based FMs [19], which allow one to define both group and feature cardinalities in an expressive way. This is even the case for alternatives to FMs, like decision models [24], which are basically tables where each row specifies a variation point, including some properties like allowable range of values, binding time, priorities, and constraints among variabilities. By taking variation points names as feature names, we can similarly extract the set of all valid configurations specified by such a model. We do not formalize this, nor the other alternative notations, for scope reasons, but we give some more detail in Section 3. Our focus is to introduce our refinement theory, instantiate it with a single FM and CK notation, and then propose and evaluate refinement templates specific to the chosen notations.

2.2. Assets

In a PL, we specify and implement features with reusable assets. So, besides a FM language, we must consider different languages for specifying and implementing assets such as requirements documents, design models, code, templates, tests, image files, XML files, and so on. For simplicity, here we focus on code assets as they are equivalent to the other kinds of assets with respect to our interests on PL refinement. Moreover, we do not commit ourselves to a specific programming language, but present concepts that apply to representative ones. So we do not present abstract syntax and semantics as previously, we just assume a notion of refinement between assets. The important issue here is not the nature of the assets contents, but actually how they are compared and referred to in a PL. We first discuss the asset comparison issue.

2.2.1. Comparison

For defining PL refinement we rely on a notion of asset refinement, which essentially is a means of comparing assets with respect to behavior preservation. As we focus on code, we first discuss existing refinement definitions for sequential programs [25,10].

Definition 2 (*Program Refinement*)**.** For programs p_1 and p_2, p_2 refines p_1, denoted by

$$p_1 \sqsubseteq p_2$$

when p_2 is at least as good as p_1 in the sense that it will meet every purpose and satisfies every input–output specification satisfied by p_1. We say that p_2 preserves the (observable) behavior of p_1.

Note that we don't require p_2 to have the same behavior as p_1; it might, for example, reduce p_1's nondeterminism. So refinement relations are pre-orders: they are reflexive and transitive. They often are partial-orders, being anti-symmetric too, but we do not require that for the just introduced relation nor for the others discussed in the remaining of the text.

For object-oriented programs, we have to deal with class declarations and method calls, which are inherently context-dependent; for example, to understand the meaning of a class declaration we must understand the meaning of its superclass. Thus, to address context issues, we make declarations explicit when dealing with object-oriented programs: '$cds \bullet m$' represents a program formed by a set of class declarations cds and a command m, which corresponds to the `main` method in Java like languages. We can then express refinement of class declarations as program refinement. In the following, juxtaposition of sets of class declarations represents their union.

Definition 3 (*Contextual Class Declaration Refinement*)**.** For sets of class declarations cds_1 and cds_2, cds_2 refines cds_1, denoted by

$$cds_1 \sqsubseteq_{cds,m} cds_2$$

in a context cds of "auxiliary" class declarations for cds_1 and cds_2, and a main command m, when

$$cds_1\ cds \bullet m \sqsubseteq cds_2\ cds \bullet m.$$

For asserting class declaration refinement independently of context, we have to prove refinement for arbitrary cds and m that form valid programs when separately combined with cds_1 and cds_2.

Definition 4 (*Class Declaration Refinement*)**.** For sets of class declarations cds_1 and cds_2, cds_2 refines cds_1, denoted by

$$cds_1 \sqsubseteq cds_2$$

when, for all commands m and sets of class declarations cds, if $cds_1\ cds \bullet m$ is well-typed then $cds_2\ cds \bullet m$ is also well-typed and

$$cds_1 \sqsubseteq_{cds,m} cds_2.$$

This definition captures the notion of behavior preservation for classes. We could also have a similar definition of refinement for aspects [26] and other kinds of code artifacts. We do not give further details because our aim here is just to introduce basic concepts useful to understand our PL refinement theory. In fact, our theory is language independent as long as the language has a notion of asset refinement such as the one just discussed. Later we show that even the PL refinement templates, which depend on specific FM and CK languages, are programming language independent. Moreover, the theory supports refinement notions with precision levels ranging from the more rigorous ones illustrated here down to notions relying only on testing. The more precise the asset refinement notion, the more precise the PL refinement notion.

2.2.2. Asset mapping

Having dealt with asset comparison, we now approach the asset reference issue. To enable unambiguous references to assets, instead of considering that a PL contains a set of assets, we assume it contains a mapping such as the following

$$
\begin{array}{l}
\{\text{Main 1} \mapsto \begin{array}{l}\texttt{class Main \{}\\ \quad\texttt{...new StartUp(...);...}\\ \texttt{\}}\end{array}\\[2ex]
\;\text{Main 2} \mapsto \begin{array}{l}\texttt{class Main \{}\\ \quad\texttt{...new OnDemand(...);...}\\ \texttt{\}}\end{array}\\[2ex]
\;\text{Common.java} \mapsto \begin{array}{l}\texttt{class Common \{}\\ \quad\texttt{...}\\ \texttt{\}}\end{array}\\
\qquad\qquad\vdots\\
\}
\end{array}
$$

from asset names referenced in other parts of the PL specification to actual assets. Such an asset mapping (AM) basically corresponds to an environment of asset declarations. This allows conflicting assets in a PL, like assets that implement alternative features, such as both `Main` classes in the illustrated AM.

2.3. Configuration knowledge

As discussed in Section 2.1, features are groups of requirements, so they must be related to the assets that realize them. The CK specifies this relation. We can express the CK in different ways. Here, abstracting some details of previous work [22], we consider that a CK is a relation from feature expressions to sets of asset names. We use propositional formulae having feature names as atoms to represent feature expressions that represent presence conditions [27]. For example, explicitly 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` aspect [26], among other assets omitted in the fifth row, should be part of the final product. Essentially, this PL uses the `AppMenu` aspect as a variability implementation mechanism [28,5] that has the effect of presenting the left screenshot in Fig. 1.[2] For usability issues, this screen should not appear in products that have only one of the Media features. This is precisely what the fifth row, in the simplified Mobile Media CK, specifies. Similarly, the Photo and Music implementations share some assets, so we write the fourth row to avoid repeating the asset names on the second and third rows.

Given a valid product configuration, CK evaluation yields the assets that constitute the corresponding product. In our example, the configuration {Photo, 240x320}[3] leads to the actual assets associated with the following names

$$\{\texttt{MM.java}, \ldots, \texttt{Photo.java}, \ldots, \texttt{Common.aj}, \ldots\}.$$

[2] We could use other variability implementation mechanisms, but that is not really needed to explain the CK concept, nor our refinement notion, which does not depend on the type of assets used by PLs.

[3] Remember we omit mandatory features for brevity.

This gives the basic intuition for formalizing the semantics of the CK notation just introduced and used in the refinement transformation templates that we show later.

In our PVS encoding, a CK is a set of items, that is, elements of a record type defined in terms of a feature expression and a set of asset names. Feature expressions are represented by propositional formulae exactly as in the FM language illustrated before. In fact, the *Formula* type is shared by the FM and CK specifications in PVS. 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.

Definition 5 (*Configuration Knowledge*).

$$\begin{aligned}
&Item : <\ exp : Formula, names : \mathcal{P}[AssetName]\ > \\
&CK : \mathcal{F}[Item] \\
&eval(K : CK, c : Configuration) : \mathcal{F}[AssetName] = \\
&\quad \{an \mid \exists i \in K \cdot satisfies(exp(i), c) \wedge an \in assets(i)\} \\
&semantics(K : CK, A : AM, c : Configuration) : \mathcal{F}[Asset] = \\
&\quad A\langle eval(K, c)\rangle.
\end{aligned}$$

We use the notation $A\langle\_\rangle$ for the relational image [29] of an asset mapping A, and $\exists i \in K \cdot p(i)$ as an abbreviation for the PVS notation $\exists i : Item \cdot i \in K \wedge p(i)$.

As for FMs, we could have used other explicit notations for expressing CK [22,30], including an extension of the one formalized here for relating feature expressions to arbitrary transformations (or tasks). Alternative approaches do not even rely on an explicit representation [31,32]. For example, some approaches assume that a *module* named after a feature implements that feature. In such cases, and in conditional compilation PLs, the CK is implicit, features relate to modules or parts of the code by their names. However, in all of these approaches, there are ways to generate a specific product according to a valid configuration, similar to what the *semantics* function specifies in Definition 5. As long as such a function exists, we can instantiate our refinement theory with the corresponding CK approach. For scope reasons, here we instantiate our theory only with the CK notation just formalized, and propose PL refinement templates specific to the same notation.

3. A theory of product line refinement

Using the concepts introduced in the previous section, we can now present our theory of PL refinement. As mentioned before, our theory does not rely on the specific languages and semantic encodings previously described. So, through explicit assumptions and axioms, we first define interfaces between our theory and the different languages that we can use to create PL artifacts. We also illustrate how these interfaces relate to the specific languages formalized in the previous section. After that, we formalize and explain our notion of PL refinement and establish associated properties that justify stepwise and compositional PL evolution.

3.1. Feature models

As discussed in Section 2.1, the set of all valid configurations often represents the semantics of a FM or alternative artifacts such as decision models. However, as different languages might express feature constraints and configurations in different ways, our PL refinement theory abstracts the details and just assumes a generic function $[\![\_]\!]$ for obtaining the semantics of a FM, or alternative artifacts, as a set of configurations. Hereafter, for making the discussion less abstract, we use FM terminology.

Assumption 1 (*Feature Model Semantics*).

$$\begin{aligned}
&FeatureModel : TYPE \\
&Configuration : TYPE \\
&[\![\_]\!] : FeatureModel \to \mathcal{P}[Configuration].
\end{aligned}$$

By specifying FMs and configuration as uninterpreted types, we assume no constraints on how specific notations might instantiate that.

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

Definition 6 (*Feature Model Equivalence*). Feature models F and F' are equivalent, denoted $F \cong F'$, whenever $[\![F]\!] = [\![F']\!]$.

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

Theorem 1 (*Feature Model Equivalence*).

$$\begin{array}{l}\forall F : \mathit{FeatureModel} \cdot F \cong F\\ \forall F, F' : \mathit{FeatureModel} \cdot F \cong F' \Rightarrow F' \cong F\\ \forall F, F', F'' : \mathit{FeatureModel} \cdot F \cong F' \wedge F' \cong F'' \Rightarrow F \cong F''.\end{array}$$

Proof. Directly from Definition 6 and the reflexivity, symmetry, and transitivity of the equality of configuration sets. □

These properties justify safe stepwise evolution of FMs, as illustrated in previous work [17], which proposes transformation templates for FMs. Based on the compositional results that we show later, these templates could be used to reason about PLs as a whole, by separately evolving the FM and not changing the CK and AM.

The concepts in Assumption 1 are all we require about FMs. With them, we can define our PL refinement notion and derive its properties. So our theory applies for any notation that describes (possibly infinite) sets of product configurations [18,19,21,20,24], regardless of how these configurations are represented, since we define *Configuration* as an uninterpreted type. For example, we can use cardinality-based FMs [19], whose configurations contain the tree structure to allow cloning. We can also use standard FODA notation [11], whose configurations are flat sets of feature names, as in Section 2.1. In fact, by instantiating the types in Assumption 1 with types *Configuration* and *WFM* in Definition 1, we can see that the *semantics* function is a possible instantiation for $[\![\_]\!]$. This binding is what allows us to later use our refinement theory to derive PL templates that use the FM notation we introduce in Section 2.1.

We actually perform that binding using the theory interpretation mechanism of PVS [16]. We import the general theory for PL refinement, providing interpretations for the uninterpreted types. If any, axioms defined in the general theory become proof obligations to ensure consistency of the specification.

3.2. Assets

Besides a precise notion of semantics for FMs or similar artifacts, for defining PL refinement we rely on means of comparing assets with respect to behavior preservation. We assume the relation $\sqsubseteq$ denotes refinement of an asset set, as, for example, shown for class declarations in Section 2.2.1. Moreover, we rely on a wf function specifying well-formedness of asset sets; this corresponds to the well known notion of type checking in most languages.

Assumption 2 (*Asset Refinement*).

$$\begin{array}{l}\mathit{Asset} : \mathit{TYPE}\\ \sqsubseteq : \mathcal{P}[\mathit{Asset}], \mathcal{P}[\mathit{Asset}] \rightarrow \mathit{bool}\\ wf : \mathcal{P}[\mathit{Asset}] \rightarrow \mathit{bool}.\end{array}$$

For brevity, we use $a \sqsubseteq a'$, for assets a and a', as an abbreviation for $\{a\} \sqsubseteq \{a'\}$. We also use $a\, a'$ as an abbreviation for $\{a, a'\}$.

Our PL refinement theory applies for any asset language with these notions as long as they satisfy the following properties, stating that asset set refinement is a pre-order.

Axiom 1 (*Asset Set Refinement is Pre-order*).

$$\begin{array}{l}\forall a : \mathcal{P}[\mathit{Asset}] \cdot a \sqsubseteq a\\ \forall a, b, c : \mathcal{P}[\mathit{Asset}] \cdot a \sqsubseteq b \wedge b \sqsubseteq c \Rightarrow a \sqsubseteq c.\end{array}$$

These are usually properties of any refinement notion because they are essential to support stepwise refinement. This is, for example, the case of existing refinement notions for object-oriented programming and modeling [9,33,34], as partly discussed in Section 2.2.1.

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

Axiom 2 (*Asset Set Refinement Compositionality*).

$$\begin{array}{l}\forall a, a' : \mathcal{P}[\mathit{Asset}] \cdot \forall s : \mathcal{P}[\mathit{Asset}] \cdot\\ \qquad a \sqsubseteq a' \wedge wf(a \cup s)\\ \quad \Rightarrow wf(a' \cup s) \wedge a \cup s \sqsubseteq a' \cup s.\end{array}$$

Such a compositionality property is essential to guarantee independent development of assets in a PL. The class refinement notion illustrated in Section 2.2.1, for example, supports it.

In general, we do not have to limit ourselves to code assets, and consider any kind of asset that supports the concepts and axioms discussed in this section. They are actually our interface to the languages used to create PL assets. Moreover, as the refinement transformation templates proposed here are programming language independent, we can prove their soundness without having to instantiate the concepts specified in this section for an specific language. Thus, we provide no binding information here.

3.3. Asset mapping

For AMs, we provide a concrete formalization, instead of defining an abstract interface. Nevertheless, there is basically no loss of generality because we only use AMs to group assets, the more relevant concept in our theory. Formally, we specify AMs in PVS as a finite function from asset names to assets, using a parameterized theory. This mapping theory defines the *AM* type, basic functions, and corresponding properties such as the fact that asset names only map to a single asset. Although we limit ourselves to a finite asset base, we can still express a possibly infinite number of products. This is often needed when considering feature cardinality and attributes, and could be expressed, for example, by having assets that are parameterized code units that can be instantiated and composed in a possibly infinite number of ways. This requires a more elaborate CK notation than the one illustrated so far, as detailed in the next section.

For reasoning about AMs, we define a notion of AM refinement. We could also define AM equivalence, but we choose the weaker refinement notion since it gives us more flexibility when evolving AMs independently of other PL elements such as FMs. As shall be clear later, we can require only refinement for AMs but not for the other elements; that is why we define equivalences for them in other sections. For AM refinement, we should map exactly the same names, not necessarily to the same assets, but to compatible assets accordingly to the notion of asset set refinement.

Definition 7 (*Asset Mapping Refinement*)**.** For asset mappings A and A', the second refines the first, denoted

$$A \sqsubseteq A'$$

whenever

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

We use $\forall n \in dom(A) \cdot p(n)$ to abbreviate the PVS notation $\forall n : AssetName \cdot n \in dom(A) \Rightarrow p(n)$. We now establish that AM refinement is a pre-order.

Theorem 2 (*Asset Mapping Refinement Pre-order*)**.**

$$\begin{array}{l} \forall A : AM \cdot A \sqsubseteq A \\ \forall A, A', A'' : AM \cdot A \sqsubseteq A' \wedge A' \sqsubseteq A'' \Rightarrow A \sqsubseteq A''. \end{array}$$

Proof. Directly from Definition 7 and the reflexivity and transitivity of asset refinement (see Axiom 1). □

To establish the compositionality results, we rely on an important property of AM refinement: if $A \sqsubseteq A'$ then products formed by using A assets are refined by products formed by corresponding A' assets. We present the proof of the following theorem in Appendix.

Theorem 3 (*Asset Mapping Compositionality*)**.** *For asset mapping A and A′, if*

$$A \sqsubseteq A'$$

then

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

3.4. Configuration knowledge

The last interface for our theory captures the basic intuition for CK semantics. We assume an $[\![\_]\!]$ function that maps product configurations and AMs into finite sets of assets. As each product is formed by a finite number of assets, the finiteness constraint imposes no limitation on our theory.

Assumption 3 (*CK Semantics*)**.**

$$\begin{array}{l} CK : TYPE \\ [\![\_]\!] : CK \rightarrow AM \rightarrow Configuration \rightarrow \mathcal{F}[Asset]. \end{array}$$

In previous sections we introduce the *Configuration* and *Asset* types, which are abstract; the *AM* type stands for the AM type discussed in the previous section. We represent the application of the semantics function to a configuration knowledge K, an asset mapping A, and a configuration c as $[\![K]\!]_c^A$.

Similarly to what we have done for FMs, we define an equivalence notion for CK. This is useful to reason about CK specifications. Two specifications are equivalent iff they have the same semantics. Notice that we demand equality of functions. The equivalence must hold for any A and c.

Definition 8 (*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*)**.**

$$\begin{aligned}
&\forall K : CK \cdot K \cong K\\
&\forall K, K' : CK \cdot K \cong K' \Rightarrow K' \cong K\\
&\forall K, K', K'' : CK \cdot K \cong K' \wedge K' \cong K'' \Rightarrow K \cong K''.
\end{aligned}$$

Proof. Directly from Definition 8 and the reflexivity, symmetry, and transitivity of the equality of functions. □

The CK evaluation must be compositional in the sense that refining an AM that generates valid products for a CK yields refined and valid products.

Axiom 3 (*Configuration Knowledge Evaluation Over Asset Mapping Refinement*)**.**

$$\begin{aligned}
&\forall A, A' : AM \cdot A \sqsubseteq A' \Rightarrow\\
&\quad \forall K : CK, c : Configuration \cdot\\
&\qquad\quad wf([\![K]\!]_c^A)\\
&\qquad \Rightarrow wf([\![K]\!]_c^{A'}) \wedge [\![K]\!]_c^A \sqsubseteq [\![K]\!]_c^{A'}.
\end{aligned}$$

By analyzing Assumption 3, note that we impose no constraints on how we represent a CK. We only require a semantics function that generates a finite set of assets given an AM and a configuration. So we can instantiate our theory with different CK notations. For the notation presented in Section 2.3, we define $[\![\_]\!]$ as asset selection enabled by feature expressions that represent presence conditions [27]. For a more complex CK notation that associates presence conditions to transformations on assets, the semantics function would basically be defined as an interpreter that evaluates transformations to generate products. For contexts that rely on cardinality-based FMs, we could opt for a CK notation supporting feature expressions that take into account feature cardinality and attributes. For instance, we could use an expression $\#F > 1$ to denote a presence condition that is applied when the cardinality of the given feature F is higher than 1. Similarly, an expression $F(att)$ would expose the F attribute values as att, which could be used by transformations performed for each instantiated value of the F attribute in a given configuration. This way, through code templates, parameterized assets, and generators, among others, we could generate an infinite number of products from a limited number of assets.

In particular, for deriving the PL refinement templates that we discuss later, we instantiate the type and $[\![\_]\!]$ function in Assumption 3 with the *CK* type and *semantics* function in Definition 5. We then have to discharge the corresponding instantiation of Axiom 3 to ensure that the instantiation is valid.

Theorem 5 (*Instantiation of Axiom 3*)**.** *For asset mapping A and A', if*

$$A \sqsubseteq A'$$

then

$$\begin{aligned}
&\forall K : CK, c : Configuration \cdot\\
&\qquad wf([\![K]\!]_c^A)\\
&\quad \Rightarrow wf([\![K]\!]_c^{A'}) \wedge [\![K]\!]_c^A \sqsubseteq [\![K]\!]_c^{A'}. \quad \square
\end{aligned}$$

The proof for the just mentioned instantiation appears in Appendix.

3.5. Product lines

We can now provide a precise definition for PLs. In particular, a PL consists of a FM, a CK, and an AM that jointly generate products, that is, valid asset sets in their target languages.

Definition 9 (*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 $c \in [\![F]\!]$,

$$wf([\![K]\!]_c^A).$$

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 AM entry might yield an invalid asset set due to conflicting assets, like two aspects [26] used as variability mechanisms [28,5], that introduce methods with the same signature in the same class. Here we demand PL elements to be coherent as explained [35,27,36].

Given the importance of the well-formedness property in this definition, we establish preservation properties related to the well-formedness function wf. These are used later to derive the main compositionality result. First we have that FM equivalence preserves well-formedness.

Lemma 1 (*Well-formedness Preservation Under Feature Model Equivalence*)**.** *For feature models F and F', asset mapping A, and configuration knowledge K, if*

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

then

$$\forall c \in [\![F']\!] \cdot wf([\![K]\!]_c^A).$$

Proof. We have that $F \cong F'$. By definition, this amounts to $[\![F]\!] = [\![F']\!]$. Since we have that $\forall c \in [\![F]\!] \cdot wf([\![K]\!]_c^A)$, we replace $[\![F']\!]$ with $[\![F]\!]$ and conclude the proof. □

Similarly, for CK we have the following.

Lemma 2 (*Well-formedness Preservation Under Configuration Knowledge Equivalence*)**.** *For feature model F, asset mapping A, and configuration knowledge K and K', if*

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

then

$$\forall c \in [\![F]\!] \cdot wf([\![K']\!]_c^A).$$

Proof. Using similar reasoning as in the proof of Lemma 1. □

Finally, for AMs we have that AM refinement also preserves well-formedness.

Lemma 3 (*Well-formedness Preservation Under Asset Mapping Refinement*)**.** *For feature model F, asset mapping A and A' and configuration knowledge K, if*

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

then

$$\forall c \in [\![F]\!] \cdot wf([\![K]\!]_c^{A'}).$$

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

$$A \sqsubseteq A' \wedge \forall c \in [\![F]\!] \cdot wf([\![K]\!]_c^A). \quad (1)$$

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

$$wf([\![K]\!]_c^{A'}). \quad (2)$$

By properly instantiating the assumption (1) with the just introduced c, we have

$$wf([\![K]\!]_c^A). \quad (3)$$

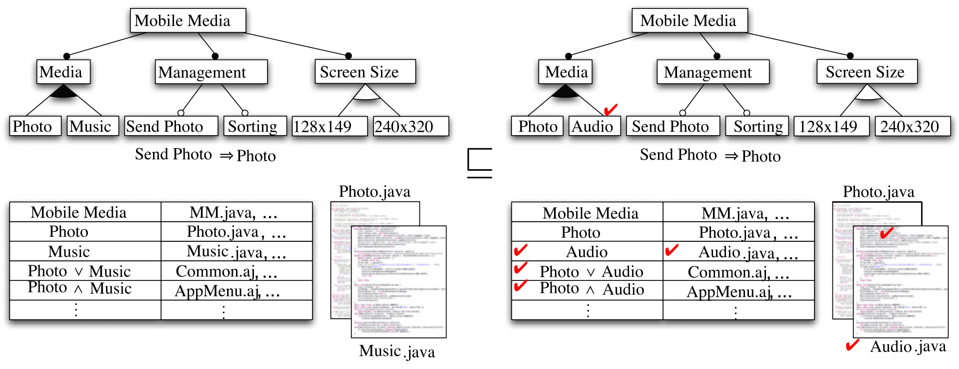
From Axiom 3, properly instantiated with A and A', and the assumption (1), we have

$$\begin{array}{l} \forall K : CK, c : Configuration \cdot \\ \quad wf([\![K]\!]_c^A) \\ \Rightarrow wf([\![K]\!]_c^{A'}) \wedge [\![K]\!]_c^A \sqsubseteq [\![K]\!]_c^{A'}. \end{array}$$

Instantiating the above with K and c, we have

$$\begin{array}{l} \quad wf([\![K]\!]_c^A) \\ \Rightarrow wf([\![K]\!]_c^{A'}) \wedge [\![K]\!]_c^A \sqsubseteq [\![K]\!]_c^{A'}. \end{array}$$

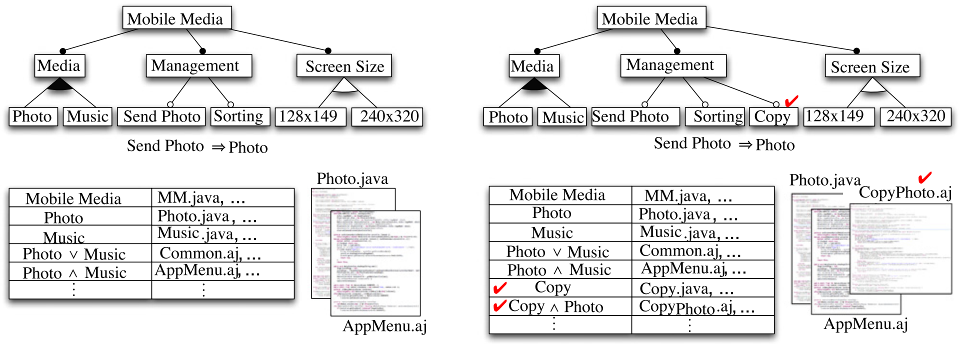
The proof (see (2)) then follows from the above and (3). □



Mobile Media	MM.java, ...
Photo	Photo.java, ...
Music	Music.java, ...
Photo ∨ Music	Common.aj, ...
Photo ∧ Music	AppMenu.aj, ...
⋮	⋮

Mobile Media	MM.java, ...
Photo	Photo.java, ...
✔ Audio	✔ Audio.java, ...
✔ Photo ∨ Audio	Common.aj, ...
✔ Photo ∧ Audio	AppMenu.aj, ...
⋮	⋮

Fig. 3. Product line renaming refinement.



Mobile Media	MM.java, ...
Photo	Photo.java, ...
Music	Music.java, ...
Photo ∨ Music	Common.aj, ...
Photo ∧ Music	AppMenu.aj, ...
⋮	⋮

Mobile Media	MM.java, ...
Photo	Photo.java, ...
Music	Music.java, ...
Photo ∨ Music	Common.aj, ...
Photo ∧ Music	AppMenu.aj, ...
✔ Copy	Copy.java, ...
✔ Copy ∧ Photo	CopyPhoto.aj, ...
⋮	⋮

Fig. 4. Adding an optional feature refinement.

3.6. *Product line refinement*

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

Similar to program and model refinement [9,33], PL refinement preserves behavior. However, it goes beyond source code and other kinds of reusable assets, and considers transformations to FMs and CK as well. Fig. 3 illustrates the case where we refine the simplified Mobile Media PL by renaming the Music feature. As indicated by check marks, this renaming requires changing the FM, CK, and AM; due to a class name change, we must apply a global renaming, so we change too the main method and other classes beyond `Music.java`.

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

3.7. *Formalization and basic properties*

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

Definition 10 (*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

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

Remember that, for a configuration c, a configuration knowledge K, and an asset mapping A related to a given PL, $[\![K]\!]_c^A$ is a well-formed set of assets. So $[\![K]\!]_c^A \sqsubseteq [\![K']\!]_{c'}^{A'}$ refers to the asset set refinement notion discussed in Section 2.2. This definition relates two PLs. Therefore, all products in the product line (F', A', K') are also well-formed. It is also important to emphasize that while AM refinement is enough to justify PL refinement, it is not a required condition for enabling it, as we see in Definition 10.

To support stepwise PL evolution, we now establish that PL refinement is a pre-order.

Theorem 6 (*Product Line Refinement Reflexivity*).

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

Proof. Let $l = (F, A, K)$ be an arbitrary PL. By Definition 10, we have to prove that

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

For an arbitrary $c \in [\![F]\!]$, just let c' be c and the proof follows from asset set refinement reflexivity (see Axiom 1). □

Theorem 7 (*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 PLs. Assume that $l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3$. By Definition 10, this amounts to

$$\forall c_1 \in [\![F_1]\!] \wedge \exists c_2 \in [\![F_2]\!] \cdot [\![K_1]\!]_{c_1}^{A_1} \sqsubseteq [\![K_2]\!]_{c_2}^{A_2} \tag{4}$$

and

$$\forall c_2 \in [\![F_2]\!] \cdot \exists c_3 \in [\![F_3]\!] \cdot [\![K_2]\!]_{c_2}^{A_2} \sqsubseteq [\![K_3]\!]_{c_3}^{A_3}. \tag{5}$$

We then have to prove that

$$\forall c_1 \in [\![F_1]\!] \cdot \exists c_3 \in [\![F_3]\!] \cdot [\![K_1]\!]_{c_1}^{A_1} \sqsubseteq [\![K_3]\!]_{c_3}^{A_3}.$$

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

$$\exists c_3 \in [\![F_3]\!] \cdot [\![K_1]\!]_{c_1}^{A_1} \sqsubseteq [\![K_3]\!]_{c_3}^{A_3}. \tag{6}$$

Properly instantiating c_1 in (4), we have

$$\exists c_2 \in [\![F_2]\!] \cdot [\![K_1]\!]_{c_1}^{A_1} \sqsubseteq [\![K_2]\!]_{c_2}^{A_2}.$$

Let c_2' be such c_2. Properly instantiating c_2' in (5), we have

$$\exists c_3 \in [\![F_3]\!] \cdot [\![K_2]\!]_{c_2'}^{A_2} \sqsubseteq [\![K_3]\!]_{c_3}^{A_3}.$$

Let c_3' be such c_3. Then we have

$$[\![K_1]\!]_{c_1}^{A_1} \sqsubseteq [\![K_2]\!]_{c_2'}^{A_2} \wedge [\![K_2]\!]_{c_2'}^{A_2} \sqsubseteq [\![K_3]\!]_{c_3'}^{A_3}.$$

By asset set refinement transitivity (see Axiom 1), we have

$$[\![K_1]\!]_{c_1}^{A_1} \sqsubseteq [\![K_3]\!]_{c_3'}^{A_3}.$$

This gives us the c_3 in (6) that completes our proof. □

As we do not require anti-symmetry for asset set refinement, we have no anti-symmetry property for PL refinement.

3.8. Examples and considerations

To explore the definition just introduced, we analyze concrete PL transformation scenarios that use specific languages for FM, CK, and assets.

3.8.1. *Feature names do not matter*

First let us see how the PL refinement definition apply to the transformation depicted by Fig. 3. The FMs 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) PL configuration {Music, 240x320} we have the target (right) PL configuration {Audio, 240x320}. As the CKs have the same structure, evaluating them with these configurations yields

$$\{\texttt{Commmon.aj}, \texttt{Music.java}, \ldots\}$$

and

$$\{\texttt{Commmon.aj}, \texttt{Audio.java}, \ldots\}.$$

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 AMs, we obtain the same assets modulo global renaming, which is a well known refinement for closed programs. This is precisely what, by Definition 10, we need for assuring that the target PL refines the source PL.

This example shows that our refinement definition focuses on the PL themselves, that is, the sets of products that we can generate. Contrasting with our previous notion of FM refactoring [17], feature names do not matter. So users will not notice they are using products from the new PL, 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.

3.8.2. *Safety for existing users only*

To further explore the PL refinement definition, let us consider now the transformation shown in Fig. 4. The target FM has an extra optional feature. So it generates all configurations of the source FM 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 FMs— configurations without Copy. As the target CK is an extension of the source CK for dealing with cases when we select Copy, 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 AMs are equal, since the target mapping is an extension of the first for names such as `CopyPhoto.java`, which appears only when we select Copy. Therefore, the resulting assets produced by each PL are the same, trivially implying program refinement and then PL refinement.

By focusing on the common configurations to both FMs, we check nothing about the new products offered by the new PL. In fact, they should be well-formed but might even not operate at all. Our refinement notion assures only that users of existing products are not disappointed by the corresponding products generated by the new PL. 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 PL without impacting existing users.

3.8.3. *Non refinements*

As discussed, the transformation depicted in Fig. 3 is a refinement. Classes and aspects [26] are transformed by a global renaming, which preserves behavior for closed programs. But suppose that, besides renaming, we change the `AppMenu.aj`[4] 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 PL refinement, consequently.

Despite not being a refinement, this menu change is a useful PL 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 refinement only apply when the intention of the transformation is to improve PL configurability or internal structure, without changing observable behavior.

3.9. *Product line refinement compositionality*

The PL refinement notion allows one to reason about a PL as a whole, considering its three elements (artifacts): FM, CK, and AM. However, for independent development of PL artifacts, we must support separate and compositional reasoning. This allows us to evolve PL artifacts independently. We first consider FMs. Replacing a FM by an equivalent one leads to a refined PL.

Theorem 8 (*Feature Model Equivalence Compositionality*). *For product lines (F, A, K) and (F', A, K), if*

$$F \cong F'$$

then

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

[4] See Section 2.3 for understanding the role this aspect plays.

Proof. For arbitrary F, F', A, K, assume that $F \cong F'$. By Definition 10, we have to prove that

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

From our assumption and Definition 6, this is equivalent to

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

For an arbitrary $c \in [\![F]\!]$, just let c' be c and the proof follows from asset set refinement reflexivity (see Axiom 1). □

We rely on FM equivalence because FM refinement, which requires $[\![F]\!] \subseteq [\![F']\!]$ instead of $[\![F]\!] = [\![F']\!]$, is not enough for ensuring that separate modifications to a FM imply PL refinement. In fact, FM refinement allows the new FM to have extra configurations that might not generate valid products; the associated FM refinement transformation would not lead to a valid PL. 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 we select both features. Refinement of the whole PL, in this case, would also demand changes to the assets and CK.

For similar reasons, to independently evolve a CK we require CK equivalence.

Theorem 9 (*Configuration Knowledge 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 8, using Definition 8 instead of Definition 6. □

Note that the reverse implication does not hold because, for example, the products that K and K' generate might differ for assets that have no impact on product behavior.[5] For similar reasons, the reverse does not hold for Theorem 8.

For AMs, we can rely only on refinement. Separately refining an AM implies refinement for the PL as a whole.

Theorem 10 (*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 (F, A, K) is a PL and that $A \sqsubseteq A'$. By Definition 10, we have to prove that

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

For an arbitrary $c \in [\![F]\!]$, let c' be such c, we then have to prove

$$[\![K]\!]^A_c \sqsubseteq [\![K]\!]^{A'}_c. \tag{7}$$

By Axiom 3 and our assumption, we have that

$$\begin{aligned} &\forall K : CK, c : Configuration \cdot \\ &\quad wf([\![K]\!]^A_c) \\ &\Rightarrow wf([\![K]\!]^{A'}_c) \wedge [\![K]\!]^A_c \sqsubseteq [\![K]\!]^{A'}_c. \end{aligned} \tag{8}$$

By properly instantiating K and c in (8), we obtain

$$\begin{aligned} &\quad wf([\![K]\!]^A_c) \\ &\Rightarrow wf([\![K]\!]^{A'}_c) \wedge [\![K]\!]^A_c \sqsubseteq [\![K]\!]^{A'}_c. \end{aligned} \tag{9}$$

From Definition 9, we have that $wf([\![K]\!]^A_c)$ for all $c \in [\![F]\!]$. Therefore, from this and (9) we obtain

$$wf([\![K]\!]^{A'}_c) \wedge [\![K]\!]^A_c \sqsubseteq [\![K]\!]^{A'}_c$$

concluding the proof (see (7)). □

Finally, we have the full compositionality theorem, which justifies the independent development of PL artifacts.

[5] Obviously an anomaly, but still possible.

Theorem 11 (*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 1, the fact that (F, A, K) is a PL, and Definition 9, we have that (F', A, K) is a PL. Then, using Theorem 8, we have

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

Similarly, from our assumptions, deductions, and Lemma 2 we have that (F', A, K') is a PL. Using Theorem 9, we have

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

Again, from our assumptions, deductions, and Lemma 3, we have that (F', A', K') is a PL. Using Theorem 10, we have

$$(F', A, K') \sqsubseteq (F', A', K'). \tag{12}$$

The proof then follows from (10)–(12), and PL refinement transitivity (see Theorem 7). □

4. Product line refinement transformations

With the refinement notion and properties introduced so far, we are able to evolve a PL by simply improving its design or by adding new products while preserving existing ones. Nevertheless, it is useful to provide guidance to developers, so they do not need to reason directly about the definitions when evolving PLs. So, in this section we illustrate different kinds of PL refinement transformation templates[6] that, for example, allow us to split an asset and add a new optional feature. We focus on templates that are programming language independent, but are specific to the FM and CK languages introduced in Section 2.

Although each transformation focuses on small changes to a PL, we can derive more elaborate refinement transformations by composing the ones proposed here. For each transformation we discuss its general intuition and motivation, and then show how we encode it and prove its soundness in the instantiation of our theory detailed by the FM and CK bindings discussed in Section 3. We present transformations that are both useful in practice and exercise different aspects of our theory by changing the PL artifacts, both in isolation and in an integrated way.

4.1. Replace feature expression

The first transformation establishes that a feature expression in a CK might be replaced for an equivalent one according to FM constraints. This may be useful to improve CK readability. Fig. 5 illustrates the abstract transformation template. We replace feature expression e with e'. The condition states that we can apply this transformation when, according to feature model F, e is equivalent to e' and e' only references features from F. As we do not change the FM and AM, we simply use F and A to refer to them. When expressions are equivalent by propositional reasoning only, we have a special case of this transformation.

A template consists of a left hand side (LHS) PL pattern and a right hand side (RHS) pattern. Each pattern refers to meta-variables that abstractly represent PL elements. For instance, in Fig. 5, we use e to denote an arbitrary feature expression and n to denote an arbitrary asset name. Similarly *its* represent other CK items that remain unchanged as they appear in both patterns. Besides the patterns, each transformation may have a pre-condition that appears below the refinement symbol. We can only apply a transformation when a concrete PL matches the LHS template and the pre-condition is valid for that matching. For simplicity, we assume that we can only apply the transformations for PLs that are valid, accordingly to Definition 9, and satisfy further constraints that we detail next. With these conditions, for each template we guarantee that the resulting PL is valid and refines the original one.

When detailed, the proposed transformation templates use the specific FM and CK languages formalized in Section 2. For the transformation in Fig. 5, the AM is not detailed nor mentioned in the pre-condition, so the template only depends on the formalized FM and CK languages. In fact, we could use this template for any FM language that allows comparing feature expressions. But, as we are working with concrete formalizations of two specific languages, we prefer not to explore this and other possible generalizations in the templates we discuss later.

We encode templates in our theory defining predicates to represent the syntactic relationships for source and target PLs (*syntax*) and transformation pre-conditions (*conditions*). In general, we map each template meta-variable and abstractions

[6] Or only transformations.

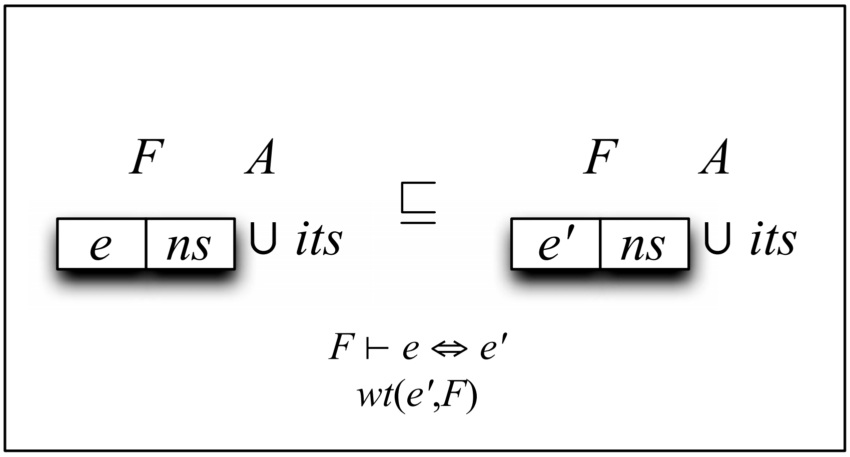


Fig. 5. Replace feature expression refinement template.

of PL elements to corresponding PVS variables with associated constraints. Therefore, for each transformation, we encode its soundness theorem as follows:

$$\begin{aligned}&\forall F, A, K, F', A', K' \ldots \cdot\\&\quad wfPL(F, A, K) \wedge syntax(\ldots) \wedge conditions(\ldots)\\&\qquad \Rightarrow (F, A, K) \sqsubseteq (F', A', K') \wedge wfPL(F', A', K').\end{aligned}$$

We parameterize the *syntax* and *conditions* predicates by the meta-variables, accordingly to the template. So, for each transformation, besides proving that its application is a PL refinement (see Definition 10), we also prove well-formedness of the resulting PL. We separate these concerns to help understanding and reasoning. For each transformation, we overload the *syntax* and *conditions* predicates with the specific transformation details. Well-formedness of the resulting PL consists of the well-formedness constraint that comes from the general theory (see Definition 9 and the first part of the conjunction presented next) plus additional constraints for the specific FM and CK languages that instantiate the refinement theory: all feature expressions in the CK refer only to features in the FM, and all asset names that appear in the CK are in the domain of the AM. These extra conditions abbreviate the transformations, by avoiding repetition in the transformation pre-conditions. The specification is as follows.

Definition 11 (*Well-formed 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 well-formed PL, denoted by $wfPL(F, A, K)$, when

$$\begin{aligned}&\forall c \in [\![F]\!] \cdot wf([\![K]\!]_c^A) \wedge\\&\forall exp \in exps(K) \cdot wt(exp, F) \wedge\\&\forall c \in [\![F]\!] \cdot eval(K, c) \subseteq dom(A).\end{aligned}$$

We prefer to use the general refinement theory notation instead of the notation of its specific instantiation. For example, we use $[\![K]\!]_c^A$ for CK evaluation, knowing that this actually stands for its instantiation $semantics(K, A, c)$, defined in our specific CK theory (see Definition 5). For functions not used in the instantiation, we adopt the specific theory notation, as in $eval(K, c)$, which performs CK evaluation against a configuration and yields the asset names that we use to build a product. In the predicate, the first part conforms to the general definition of PLs (see Definition 9), which states that all products must be well-formed. The other constraints are specific to the CK language we detail in Section 2.3. The wt predicate, discussed in Section 2.1, asserts that a formula only contains names from the given FM. Finally, the *exps* predicate yields the set of feature expressions in a given CK.

The remainder of this section details the *syntax* and *conditions* predicates that encode the transformation, lemmas that better structure the proof, and the refinement and well-formedness proofs.

4.1.1. Syntax and conditions

The syntactic similarities and differences between the source and target PLs establish the *syntax* predicate. Both F and A are not detailed in the template, so we do not need to mention them in this predicate. By Fig. 5, we know that the source and target CKs differ only with respect to one row. We encode each of these rows as a CK item, i_1 for the source CK and i_2 for the target CK. All other CK items (*its*) are the same. Therefore, we express the source CK as the union of i_1 and *its*. We encode the target CK similarly. To express that the asset names referred to in i_1 and i_2 are the same (n in the template), we use the *names* function, which yields the asset names to which a given CK item refers to.

$$\begin{aligned}&syntax(K, K', i_1, i_2, its) =\\&\quad K = \{i_1\} \cup its \wedge\\&\quad K' = \{i_2\} \cup its \wedge\\&\quad names(i_1) = names(i_2).\end{aligned} \tag{13}$$

The conditions establish the relation between e and e', that is, the feature expressions in i_1 and i_2. We specify that all product configurations from a feature model F lead to equivalent evaluation for the feature expressions in both i_1 and i_2. We also specify that the feature expression in i_2 only references names from F. The *exp* function yields the feature expression of a given CK item.

$$\begin{aligned}&conditions(F, i_1, i_2) =\\&\quad \forall c \in [\![F]\!] \cdot satisfies(exp(i_1), c) \Leftrightarrow satisfies(exp(i_2), c) \wedge\\&\quad wt(exp(i_2), F).\end{aligned} \tag{14}$$

4.1.2. Lemmas

Given the specifications for the *syntax* and *conditions* predicates, we now proceed to prove soundness of the Replace Feature Expression transformation. To simplify the proof, we rely on a lemma that captures the fact that replacing a feature expression for an equivalent one, according to the FM, does not affect CK evaluation. Since the expressions are equivalent, for each product configuration, their evaluation yield the same result, thus, all products remain the same. We use the general theory notation to express FM and CK semantics, assuming the binding with our specific languages for FM and CK. Therefore $[\![F]\!]$ and $[\![K]\!]_c^A$ respectively refer to the *semantics* functions in Definitions 1 and 5.

Lemma 4 (*Replace Feature Expression Results in Equal CK Evaluation*)**.** *For product line* (F, A, K)*, configuration knowledge* K'*, CK items* i_1*,* i_2*, and the set of CK items its, if*

$$syntax(K, K', i_1, i_2, its) \wedge conditions(F, i_1, i_2)$$

then

$$\forall c \in [\![F]\!] \cdot [\![K]\!]_c^A = [\![K']\!]_c^A. \quad \square$$

For space reasons, the proof of this lemma and other lemmas used in the subsequent transformations are available in the online appendix, together with the PVS encoding for all theories.

4.1.3. Refinement

In the transformation illustrated in Fig. 5, we replace a CK feature expression for an equivalent one, according to the FM. The general theorem for proving soundness establishes that we have to prove both PL refinement and well-formedness of the resulting PL. In what follows, we prove that this transformation is a PL refinement.

Proof. For arbitrary F, A, K, K', i_1, i_2, *its*, assume the *syntax* and *conditions* predicates previously illustrated (see Predicates (13) and (14)), and $wfPL(F, A, K)$. By Definition 10, we have to prove that

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

As mentioned, we use the general theory notation $[\![F]\!]$ to represent our specific function $semantics(F)$. The same applies to other functions in this proof and the subsequent ones. For an arbitrary $c \in [\![F]\!]$, let c' be c and we have then to prove that

$$[\![K]\!]_c^A \sqsubseteq [\![K']\!]_c^A.$$

By Lemma 4 and the assumptions, properly instantiated with the variables just introduced, we have that

$$\forall c \in [\![F]\!] \cdot [\![K]\!]_c^A = [\![K']\!]_c^A.$$

The proof follows by instantiating this with the c used above and from asset set refinement reflexivity (see Axiom 1). $\square$

4.1.4. Well-formedness of the resulting PL

As explained earlier in this section, the general soundness theorem also includes proving well-formedness of the resulting PL (see Definition 11).

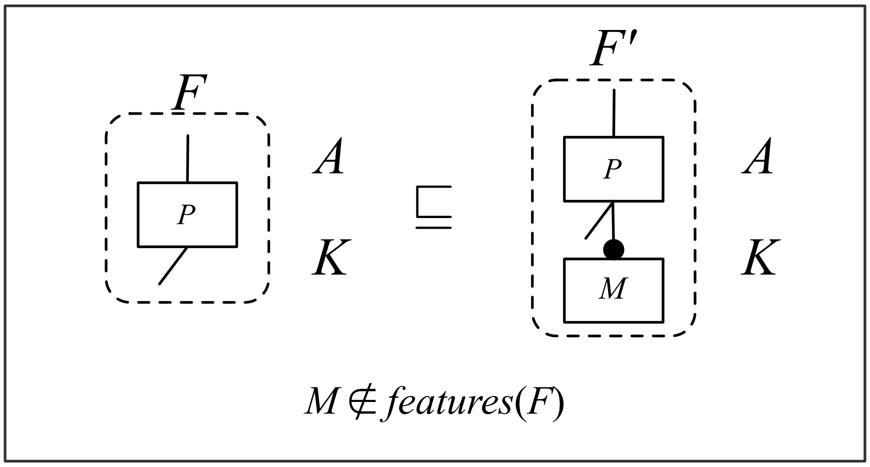


Fig. 6. Add mandatory feature refinement template.

Proof. For arbitrary $F, A, K, K', i_1, i_2, its$, assume $wfPL(F, A, K)$ and the *syntax* and *conditions* predicates previously illustrated (see Predicates (13) and (14)). By Definition 11, we have to prove that

$$\begin{aligned}
&\forall c \in [\![F]\!] \cdot wf([\![K']\!]_c^A) \wedge\\
&\forall exp \in exps(K') \cdot wt(exp, F) \wedge\\
&\forall c \in [\![F]\!] \cdot eval(K', c) \subseteq dom(A).
\end{aligned}$$

In the previous section, using Lemma 4, we have obtained $[\![K]\!]_c^A = [\![K']\!]_c^A$ for any $c \in [\![F]\!]$. Following the same steps, by substitution and the fact that $wfPL(F, A, K)$, we prove the first predicate in the conjunction.

We also have to prove that the feature expressions in K' only refer to feature names in F. From $wfPL(F, A, K)$ and Definition 11, we have that all feature expressions in K refer to features in F. From this and $K = \{i_1\} \cup its$, we know that all feature expressions in *its* only contain names in F. We then obtain what we want by observing that $K' = \{i_2\} \cup its$ and that the *conditions* predicate gives us $wt(exp(i_2), F)$. Finally, asset names in K' are the same as in K, thus we can directly prove that all names referred in K' are in the domain of A. □

4.2. Add mandatory feature

As an intermediate step when evolving a PL, it might be useful to introduce new features to the FM, for better organization and understanding or simply to extend the PL with new functionalities. For example, a report application might have only one fixed output method, say HTML, referred in the FM as Output. We can add a new mandatory subfeature HTML, and then later create new features that will form an or relation with HTML. The template in Fig. 6 establishes that it is possible to add a mandatory feature to the FM, without changing A and K. In the template we use the meta-variables M and P to denote features, whereas F denotes the original FM. We add a new feature M as the child of an existing feature P.

4.2.1. Syntax and conditions

Since we are adding a new feature M as a child of P, we declare that P is an existing feature. We obtain the resulting FM (F') by adding the new feature and a new formula expressing the mandatory relation between P and M. We establish the *syntax* predicate for this transformation as follows.

$$\begin{aligned}
&syntax(F, F', P, M) =\\
&\quad features(F') = features(F) \cup \{M\} \wedge\\
&\quad formulae(F') = formulae(F) \cup \{P \Leftrightarrow M\} \wedge\\
&\quad P \in features(F)
\end{aligned} \tag{15}$$

The condition required for this transformation specifies that M is not in the original FM.

$$\begin{aligned}
&conditions(F, M) =\\
&\quad M \notin features(F).
\end{aligned} \tag{16}$$

4.2.2. Lemmas

We use two lemmas to simplify the soundness proof of this transformation. The first states that adding a mandatory node, the child of P, to a FM preserves configurations that do not include P. Moreover, configurations that include P are simply extended with M. This holds because the added feature is a mandatory child of P, therefore, when we include P, we also have to include M in the resulting configuration. The resulting FM is well-formed since the formula we add only references P, which was an existing feature, and M, which is the new feature we add.

Lemma 5 (*Add Mandatory Node to Feature Model*). *For feature models F and F′, feature names P and M, if*

$$syntax(F, F', P, M) \wedge conditions(F, M)$$

then

$$\forall c \in [\![F]\!] \cdot IF\ (P \in c)\ (c \cup \{M\} \in [\![F']\!])\ ELSE\ (c \in [\![F']\!]) \wedge wfFM(F'). \quad \Box$$

The second lemma states that CK evaluation is not affected by extending valid configurations with features not in the FM. This is true because all feature expressions in K only refer to feature names from F, since they are part of a well-formed PL. Thus, including the M feature in the configuration does not affect CK evaluation.

Lemma 6 (*Dead Feature Does Not Affect CK Evaluation*). *For product line* (F, A, K) *and feature name M, if*

$$wfPL(F, A, K) \wedge conditions(F, M)$$

then

$$\forall c \in [\![F]\!] \cdot [\![K]\!]^A_c = [\![K]\!]^A_{c \cup \{M\}}. \quad \Box$$

4.2.3. Refinement

Fig. 6 illustrates the inclusion of a mandatory feature. As A and K remain the same, products generated by both source and target PLs are the same, as we show in the following.

Proof. For arbitrary F, A, K, F', P, M, assume the *syntax* and *conditions* predicates previously illustrated (see Predicates (15) and (16)). By Definition 10, we have to prove that

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

For an arbitrary $c \in [\![F]\!]$, by Lemma 5, we have

$$IF\ (P \in c)\ (c \cup \{M\} \in [\![F']\!])\ ELSE\ (c \in [\![F']\!]).$$

By case analysis, consider that $P \in c$, we then have $c \cup \{M\} \in [\![F']\!]$. Properly instantiating this in c' we have to prove that

$$[\![K]\!]^A_c \sqsubseteq [\![K]\!]^A_{c \cup \{M\}}.$$

By Lemma 6 and the precondition, instantiated with the variables introduced above, we have $[\![K]\!]^A_c = [\![K]\!]^A_{c \cup \{M\}}$. The first case of the proof then follows from this and asset set refinement reflexivity (see Axiom 1).

Now consider that $P \notin c$. In this case, we have that $c \in [\![F']\!]$. The proof then follows by taking c' as c and applying asset set refinement reflexivity (see Axiom 1). $\Box$

4.2.4. Well-formedness of the resulting PL

We also prove well-formedness of the resulting PL.

Proof. For arbitrary F, A, K, F', P, M, assume $wfPL(F, A, K)$ and the *syntax* predicate previously illustrated (see Predicate (15)). By Definition 11, we have to prove that

$$\begin{aligned}
&\forall c \in [\![F']\!] \cdot wf([\![K]\!]^A_c) \wedge \\
&\forall exp \in exps(K) \cdot wt(exp, F') \wedge \\
&\forall c \in [\![F']\!] \cdot eval(K, c) \subseteq dom(A).
\end{aligned}$$

In the previous section, using Lemma 5 and case analysis, we show that the source and target PLs generate the same products. From that and our assumption, by Definition 11, we are able to obtain the first part of the conjunction we have to prove. We already have that feature expressions in K reference only features from F. Since $features(F) \subseteq features(F')$, this holds too for the resulting FM. As CK evaluation is not affected by the new feature, we obtain too the last part of the conjunction. $\Box$

4.3. Split assets

Sometimes it is important to extract code from an existing asset so that we improve variability or maintainability. So the following transformation considers the case where we split an asset in two. It is important to highlight that the actual way that we split the asset is not detailed in this transformation as this is asset language specific. The important thing is that this splitting results in an asset set refinement. Fig. 7 formalizes the transformation template. The target AM has the original asset name n now associated to a' and the inclusion of n', associated with the new asset a''. The CK item that referred to n now also refers to n', and the feature expression remains the same. To apply this template to a concrete PL, we have to check asset set refinement for the specific assets involved in the splitting. This can be done using either the refinement definition of the associated asset language or specific asset refinement templates. In both cases, semantic details of the specific asset language are encapsulated by the definition and the templates. For example, typical noninterference [37] properties needed when splitting assets are captured by the definition and preconditions of the asset refinement templates. This way, we can apply the split asset template to PLs that use different asset languages and require different noninterference properties.

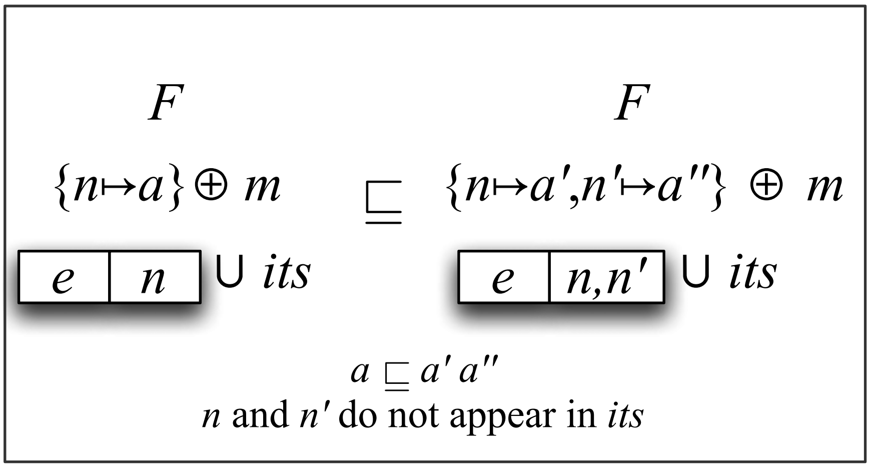


Fig. 7. Split assets refinement template.

4.3.1. Syntax and conditions

The *syntax* predicate states that asset mappings A and A' only differ with respect to the modified assets, all other mappings (m) are the same. The same happens with the CKs K and K', that only differ on i_1 and i_2, which have the same feature expression. Finally, i_2 has an extra asset name associated to it, since we create a new asset when splitting the original one.

$$\begin{aligned} & syntax(A, K, A', K', i_1, i_2, its, n, n', a, a', a'', m) = \\ & \quad A = \{n \to a\} \oplus m \wedge \\ & \quad A' = \{n \to a', n' \to a''\} \oplus m \wedge \\ & \quad K = \{i_1\} \cup its \wedge \\ & \quad K' = \{i_2\} \cup its \wedge \\ & \quad exp(i_1) = exp(i_2) \wedge \\ & \quad names(i_1) = \{n\} \wedge \\ & \quad names(i_2) = \{n, n'\}. \end{aligned} \tag{17}$$

The pre-condition requires the asset a to be refined by a' and a''. Also, the remaining CK items cannot refer to n or n', since that would result in potentially invalid programs in the target PL, as we split the asset originally referred by n in two assets.

$$\begin{aligned} & conditions(a, a', a'', n, n', its) = \\ & \quad a \sqsubseteq a' \, a'' \wedge \\ & \quad \forall\, it \in its \cdot n \notin names(it) \wedge n' \notin names(it). \end{aligned} \tag{18}$$

4.3.2. Lemmas

We use a number of lemmas to simplify the soundness proof. The first lemma we show states that CK evaluation is not affected by not activated items. This holds since the PL differs only by a single CK item that has the same feature expression. So when it is not activated in the source PL, it is not activated in the target PL, resulting in equal products.

Lemma 7 (*CK Evaluation is Not Affected When Item is Not Activated*)**.** *For product line (F, A, K), asset mapping A', configuration knowledge K', CK items i_1 and i_2, CK items set its, asset names n and n', assets a, a' and a'', and set of mappings m, if*

$$syntax(A, K, A', K', i_1, i_2, its, n, n', a, a', a'', m) \wedge conditions(a, a', a'', n, n', its)$$

then

$$\forall c \in [\![F]\!] \cdot \neg satisfies(exp(i_1), c) \Rightarrow [\![K]\!]_c^A = [\![K']\!]_c^{A'}. \quad \square$$

The second lemma states that activation of the single CK item we change leads to asset sets that are equal except for the substitution of a for a' and a''. This holds because we know from the *syntax* predicate that feature expressions are the same for i_1 and i_2, and we also know which assets these items specifically refer to.

Lemma 8 (*CK Evaluation When Item is Activated*)**.** *For product line (F, A, K), asset mapping A', configuration knowledge K', CK items i_1 and i_2, CK items set its, asset names n and n', assets a, a' and a'', and set of mappings m, if*

$$syntax(A, K, A', K', i_1, i_2, its, n, n', a, a', a'', m) \wedge conditions(a, a', a'', n, n', its)$$

then

$$\begin{aligned} & \forall c \in [\![F]\!] \cdot satisfies(exp(i_1), c) \Rightarrow \\ & \quad [\![K]\!]_c^A = \{a\} \cup [\![its]\!]_c^A \wedge [\![K']\!]_c^{A'} = \{a', a''\} \cup [\![its]\!]_c^{A'}. \quad \square \end{aligned}$$

Finally, the following lemma states that evaluation of the remaining items in the CK (*its*) is the same for both the source and target AMs. This holds since these items are the same and do not refer to n or n', as specified in the *conditions* predicate.

Lemma 9 (*CK Evaluation is the Same for Remaining CK Items*)**.** *For product line* (F, A, K)*, asset mapping* A'*, configuration knowledge* K'*, CK items* i_1 *and* i_2*, CK items set its, asset names* n *and* n'*, assets* a*,* a' *and* a''*, and set of mappings* m*, if*

$$syntax(A, K, A', K', i_1, i_2, its, n, n', a, a', a'', m) \wedge conditions(a, a', a'', n, n', its)$$

then

$$\forall c \in [\![F]\!] \Rightarrow [\![its]\!]_c^A = [\![its]\!]_c^{A'}. \quad \square$$

4.3.3. *Refinement*

In this template, we split an existing asset in two. Thus, we add a new entry to the AM and change the CK item that refers to this asset, including the new asset.

Proof. For arbitrary $F, A, K, A', K', i_1, i_2, its, n, n', a, a', a'', m$, assume the *syntax* and *conditions* predicates previously illustrated (see Predicates (17) and (18)). By Definition 10, we have to prove that

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

For an arbitrary $c \in [\![F]\!]$, let c' be c and we have to prove that

$$[\![K]\!]_c^A \sqsubseteq [\![K']\!]_c^{A'}.$$

By case analysis, now consider that $\neg satisfies(exp(i_1), c)$. By Lemma 7, instantiated with the variables just introduced, we have that

$$\forall c \in [\![F]\!] \cdot \neg satisfies(exp(i_1), c) \Rightarrow [\![K]\!]_c^A = [\![K']\!]_c^{A'}.$$

The proof follows by instantiating this with the c used before and from asset set refinement reflexivity (see Axiom 1).

Now we consider the case $satisfies(exp(i_1), c)$. By Lemma 8, instantiated with the variables introduced above, by the fact that $satisfies(exp(i_1), c)$ we have that for all $c \in [\![F]\!]$

$$[\![K]\!]_c^A = \{a\} \cup [\![its]\!]_c^A \wedge [\![K']\!]_c^{A'} = \{a', a''\} \cup [\![its]\!]_c^{A'}.$$

Therefore, we have to prove that

$$\{a\} \cup [\![its]\!]_c^A \sqsubseteq \{a', a''\} \cup [\![its]\!]_c^{A'}.$$

By Lemma 9, also instantiated with the variables above, we have that for all $c \in [\![F]\!]$

$$[\![its]\!]_c^A = [\![its]\!]_c^{A'}.$$

Given the above, we can use the asset set refinement compositionality axiom (see Axiom 2) to obtain

$$\begin{aligned} & a \sqsubseteq a'\,a'' \wedge wf(\{a\} \cup [\![its]\!]_c^A) \\ & \quad \Rightarrow wf(\{a', a''\} \cup [\![its]\!]_c^{A'}) \wedge \{a\} \cup [\![its]\!]_c^A \sqsubseteq \{a', a''\} \cup [\![its]\!]_c^{A'}. \end{aligned}$$

From the *conditions* predicate, we have that $a \sqsubseteq a'\,a''$. Since the source PL (F, A, K) is a well-formed PL, we have that $wf(\{a\} \cup [\![its]\!]_c^A)$. This concludes our proof. $\square$

4.3.4. *Well-formedness of the resulting PL*

In what follows, we detail the proof that the target PL is well-formed, by case analysis on the CK item evaluation.

Proof. For arbitrary $F, A, K, A', K', i_1, i_2, its, n, n', a, a', a'', m$, assume the *syntax* and *conditions* predicates previously illustrated (see Predicates (17) and (18)), and $wfPL(F, A, K)$. By Definition 11, we have to prove that

$$\begin{aligned} & \forall c \in [\![F]\!] \cdot wf([\![K']\!]_c^{A'}) \wedge \\ & \forall exp \in exps(K') \cdot wt(exp, F) \wedge \\ & \forall c \in [\![F]\!] \cdot eval(K', c) \subseteq dom(A'). \end{aligned}$$

By Definition 9, we have that $wf([\![K]\!]_c^A)$ for all $c \in [\![F]\!]$. So, using a similar reasoning as the one we use to prove refinement, we have that products in the source PL are either the same or similar to products in the target PL, the only difference is that a' and a'' replace a, which results in well-formed products by the asset set refinement compositionality axiom (see Axiom 2) as illustrated in the final step of the proof in the previous section. Thus, we prove the first predicate above. Feature expressions remain unchanged, and so does the FM. Thus, we prove the second part of the conjunction. Since the source PL (F, A, K) is a well-formed PL, we have that $\forall c \in [\![F]\!] \cdot eval(K, c) \subseteq dom(A)$. The only difference between A and A' is the inclusion of the new asset name n', mapped to a''. Therefore, the domain of A is a subset of the domain of A'. For K and K', the only difference in asset names is that the target CK (K') also refers to n'. Thus, we prove that asset names in K' only refer to names in the domain of A'. $\square$

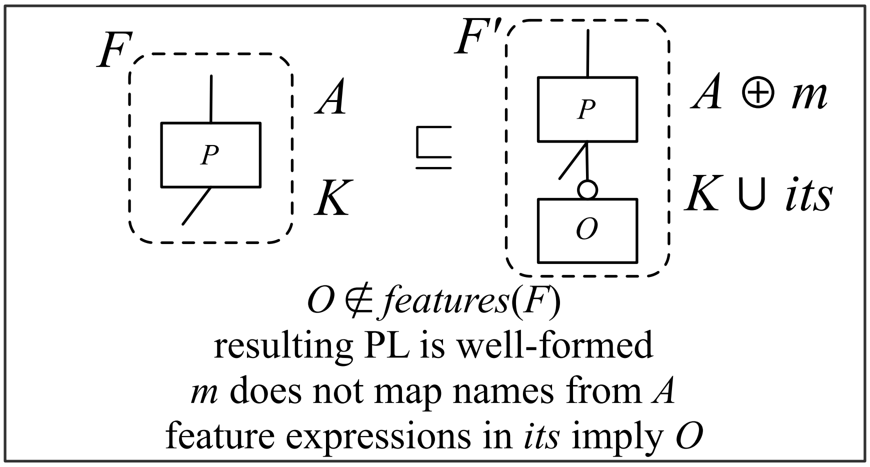

F
A
K
P
⊑
F'
P
O
A ⊕ m
K ∪ its
O ∉ features(F)
resulting PL is well-formed
m does not map names from A
feature expressions in its imply O

Fig. 8. Add optional feature refinement template.

4.4. Add optional feature

The transformation we now consider specifies that adding an optional feature is possible when the extra rows added to the original CK are only enabled by the selection of the new optional feature. This assures that products built without the new feature correspond exactly to the original PL products. Fig. 8 illustrates it. This transformation generalizes the refinement illustrated by Fig. 4, where we add the optional Copy feature to an existing PL, abstracting the details of a specific situation like that.

In this refinement transformation, we basically impose no constraints on the original PL elements; we only require the original FM to have at least one feature, identified in the transformation by the meta-variable P. We can extend the AM as wished, provided that the result is a valid AM. So in this case the new entries should not map names that are already mapped. Similarly, O should be a feature name that does not appear in the original FM, otherwise we would have an invalid FM in the target PL.

4.4.1. Syntax and conditions

We establish the *syntax* predicate for this transformation as follows. We add a new O feature as an optional node child of P, new items (its) to K, and new mappings (m) to A.

$$\begin{aligned}
&syntax(F, A, K, F', A', K', P, O, its, m) = \\
&\quad features(F') = features(F) \cup \{O\} \wedge \\
&\quad formulae(F') = formulae(F) \cup \{O \Rightarrow P\} \wedge \\
&\quad P \in features(F) \wedge \\
&\quad A' = A \oplus m \wedge \\
&\quad K' = K \cup its.
\end{aligned} \tag{19}$$

We require well-formedness of the target PL as a condition for the transformation. The new feature O cannot be an existing feature in F and new entries (m) do not override existing entries in the asset mapping A. The condition specified by this template also establishes that the new items added to the source CK are only activated by O. Thus, for any configuration, if the feature expression of the new CK items evaluate as true, O also evaluates as true against that configuration.

$$\begin{aligned}
&conditions(F, A, K', F', A', its, O, m) = \\
&\quad wfPL(F', A', K') \wedge \\
&\quad O \notin features(F) \wedge \\
&\quad \forall n \in dom(m) \cdot n \notin dom(A) \wedge \\
&\quad \forall c \cdot \forall exp \in exps(its) \cdot satisfies(exp, c) \Rightarrow satisfies(O, c).
\end{aligned} \tag{20}$$

4.4.2. Lemmas

We rely on two lemmas to simplify the soundness proof for this transformation. The first establishes that adding an optional feature O to a FM maintains previous product configurations. This holds since the feature we add is optional, therefore, we still have all previous configurations plus the new ones containing the new feature. The resulting FM is well-formed since the formula we add only references P, which was an existing feature, and O, which is the feature we add.

Lemma 10 (*Add Optional Node to Feature Model*)**.** *For feature models F, F', and feature names P and O, if*

$$\begin{aligned}&features(F') = features(F) \cup \{O\} \wedge\\&formulae(F') = formulae(F) \cup \{O \Rightarrow P\} \wedge\\&P \in features(F) \wedge O \notin features(F)\end{aligned}$$

then

$$\forall c \in [\![F]\!] \cdot c \in [\![F']\!] \wedge wfFM(F'). \quad \square$$

The other lemma we establish uses the syntax and conditions specified in Eqs. (19) and (20) to define that, for all product configurations in the source PL, the product generated by CK evaluation is the same product generated by evaluating the configuration in the target PL. This holds because the new items (*its*) are only activated by the new feature O. So, previous configurations, that do not contain O, cannot generate different products, even with the modifications performed to the CK and AM.

Lemma 11 (*CK Evaluation Not Affected by New Optional Feature with Guarded Items*)**.** *For product line (F, A, K), feature model F', asset mapping A', CK items items, features P and O, if*

$$wfPL(F, A, K) \wedge syntax(F, A, K, F', A', K', P, O, its, m) \wedge conditions(F, A, K', F', A', its, O, m)$$

then

$$\forall c \in [\![F]\!] \cdot [\![K]\!]_c^A = [\![K']\!]_c^{A'}. \quad \square$$

4.4.3. Refinement

Proof. For arbitrary F, A, K, F', A', K', P, O, *its*, assume the *syntax* and *conditions* predicates previously illustrated (see Predicates (19) and (20)), and $wfPL(F, A, K)$. By Definition 10, we have to prove that

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

For an arbitrary $c \in [\![F]\!]$, using Lemma 10 instantiated with the variables introduced above, we have that $c \in [\![F']\!]$. Let c' be c and we have to prove that

$$[\![K]\!]_c^A \sqsubseteq [\![K']\!]_c^{A'}.$$

From Lemma 11, instantiated with c, we have that

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

The proof follows by instantiating this with the c used above and from asset refinement reflexivity. $\square$

4.4.4. Well-formedness of the resulting PL

The well-formedness of the resulting PL is a template precondition. Therefore the proof is trivial. We do not know details about the new product configurations (those that include O), but the precondition guarantees that the extensions to the AM and CK lead to valid products.

4.5. Other templates

As mentioned before, we can derive other templates by composing the presented ones. We have also specified and proved variations of the presented templates. In some situations, it is useful to increase PL variability. It is then important to have templates to justify that. For example, we could define the Add Alternative Feature template using the same conditions from Add Optional Feature (see Fig. 8). In summary, we can create an alternative relation or add a new feature to a previous alternative relation, and add extra rows to the original CK for the new feature. The AM can also be modified provided that the result is a valid AM. Likewise, we could define the Add Or Feature template. In both cases, the only difference is the type of the added feature. For both templates, the proofs follow the same reasoning applied to the Add Optional Feature template. In fact, both templates have a condition guaranteeing that the products containing the new feature are well-formed. Moreover, notice that the resulting PL contains all configurations (c) of the original one. All products generated by c are the same in the original and resulting PLs, since the extra rows are not used by c, as assured by a precondition.

As another example, we could propose the Merge Assets template. Sometimes it is important to reduce the number of assets to improve maintainability. We can merge some of them. Roughly, we can visualize this template as Split Assets applied from right to left (see Fig. 7). We merge two assets a' and a'' into a. The FM remains the same, and we only change one row in the CK. The only difference is the asset set refinement condition: $a'\,a'' \sqsubseteq a$. The template's syntax is the same and the proof also follows similar reasoning as in the Split Assets template. The PL refinement definition only requires each existing product in the original PL to have a corresponding one in the target PL. Thus, if we suppose a as a new asset, in the target PL, that covers functionality of two assets a' and a'' from the original PL, and every product from the original PL that had a' and a'' now has a instead, we obtain PL refinement.

Although our templates define small-grained transformations, we can derive interesting coarse-grained transformations by composing the small-grained ones. For example, by using Add Mandatory Feature followed by Replace Feature expression, we can introduce a new feature in the PL and update the CK to associate assets to the new feature. By analyzing the evolution of two different PLs [38], we noticed that this new template, and others, are useful for safely evolving PLs.

5. Related work

Opdyke coined the term refactoring in his thesis [7]. He informally proposes refactorings as behavior-preserving program transformations that improve quality factors [39]. The definition aims to support the iterative design of object-oriented application frameworks, approaching behavior preservation through successive compilation and tests. Although Opdyke's work, and later refactoring definitions, apply to frameworks, which are often used in current PL development, and introduce variation points, in a PL we have to deal with a number of products, not a single program, and this might involve conflicting sets of class declarations. Furthermore, PL evolution, as explored here, goes beyond code assets and considers specific PL artifacts such as FMs and CK.

The idea of transforming a program into another one with compatible observable behavior is even older than refactoring. In fact, early work by Hoare and Dijkstra [13,14] formalize behavior preservation through the notion of refinement, which underlies typical refactoring notions. Those refinement notions were later used to formally derive laws of programming, which state properties about programming constructs and are useful for reasoning about programs [40]. A number of paradigms have benefited from algebraic programming laws. For example, laws of Occam [41] provide useful properties of concurrency and communication. As another example, laws of imperative programming [42] have been useful not only for providing algebraic semantic definitions but also for establishing a sound basis for formal software development methods, such as refactoring. In this work, we take into consideration specific PL characteristics and propose a refinement notion for it. Additionally, we prove a number of properties about the notion and associated transformation templates, which can be seen as a small set of PL laws.

Borba et al. [9] propose a relatively complete set of programming laws for the Refinement Object-Oriented Language (ROOL), which is a language similar to sequential Java but with copy semantics. They define this set of primitive laws (bidirectional refinement transformations) based on a refinement notion, and proved that they are behavior preserving based on a weakest preconditions semantics for ROOL [9]. By composing their laws, they formalize a number of object-oriented program refactorings. Silva et al. [43] extend the previous work considering a language with reference semantics (rCOS). They prove the correctness of each one of the laws with respect to rCOS semantics. By instantiating our theory with their refinement notions, we can jointly apply the program refactorings in ROOL and rCOS together with PLs templates such as Split Assets, which we describe here. Moreover, using the compositionality result, we could safely evolve PLs by applying ROOL and rCOS templates to the PL asset base. We could also do that to other programming and modeling languages that have similar refinement notions.

Alves et al. [17] propose an informal notion of PL refactoring and a number of FM refactoring and equivalence templates. Gheyi et al. [18] extend that by proposing a complete and minimal catalog of FM transformation templates. We go beyond that by improving and formalizing the underlying PL refinement notion, and also by considering not only FMs, but CK and assets too, both in isolation and in an integrated way. Nevertheless, using the compositionality results we present here, some of the mentioned FM templates could be used to safely evolve a PL by changing the FM only. Even earlier work [44] on PL refactoring focuses on PL 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 we can use to resolve these problems. Besides being specific to architectural assets, this work does not deal with other PL artifacts such as FMs and CK. There is also no notion of behavior preservation for PLs, as captured here by our notion of PL refinement.

A number of approaches [45–48] focus on refactoring a product into a PL, not exploring PL evolution in general, as we do here. First, Kolb et al. [45] discuss a case study in refactoring legacy code components into a PL implementation. They define a systematic process for refactoring products with the aim of obtaining PLs assets. There is no discussion about FMs and CK. Moreover, behavior preservation and configurability of the resulting PLs are only checked by testing. Similarly, Kastner et al. [48] focus only on transforming code assets, implicitly relying on refinement notions for aspect-oriented programs [10]. As discussed here and elsewhere [6] these are not adequate for justifying PL refinement and refactoring. Trujillo et al. [46] go beyond code assets, but do not explicitly consider transformations to FM and CK. 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. [47] 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 FM transformations and assumes an implicit notion of CK based on the idea of derivatives. So it does not consider explicit CK 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 we can map their technique to our notion of asset set refinement, we can use both theories together.

The notion of PL refinement discussed here first appeared in a PL refactoring tutorial [6]. Besides introducing PL refactoring, it defines a notion of population refactoring, useful for justifying PL derivation from several programs, for example. It also illustrates different kinds of refactoring transformation templates that can be useful for deriving and evolving PLs. Later on we extended [15] the initial formalization of the tutorial making clear the interfaces between our theory and languages used to describe PL artifacts such as FMs and CK. 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. In this article, we generalize the previous theory and instantiate the new theory using previously encoded theories for FM [18] and CK. Based on this instantiation, we propose PL refinement templates and prove their soundness.

Czarnecki et al. [19] introduce cardinality-based feature modeling. They specify a formal semantics for FMs and translate cardinality-based FMs into context-free grammars. They also propose FM specializations, a transformation that reduces configurability. We explain how we can instantiate our theory with their FM notation and an extension of the CK notation we explore here. In addition, we can see their formal treatment of FM specialization as the opposite of our notion of FM refactoring [17]. Schobbens et al. [21] provide a formal semantics for FMs (in their work, feature diagrams), in terms of generic formalization of syntax and semantics. This effort results in a language – Varied Feature Diagrams (VFD) – that is expressively complete, making it able to express diverse constructs. Our general PL theory declares Assumption 1, specifying a general FM semantics definition that is similar to theirs. We can express our specific FM semantics in their formalization. We can also instantiate our theory for dealing with other FM notations and semantics proposed by Schobbens et al. [21].

Thaker et al. present techniques for verifying type safety properties of AHEAD [31] PLs using FMs and SAT solvers [35]. They extract properties from feature modules and verify that they hold for all PL members. Czarnecki and Pietroszek also propose a well-formedness verification approach, but for feature-based model templates [27]. Such templates consist of an FM and an annotated model expressed in some general modeling language such as UML or a domain-specific modeling language. They also use SAT solvers, checking the FM against constraints to verify that no ill-formed template instances can be produced. Kaestner et al. propose a technique for verifying type safety of annotation-based PLs [36]. They extend the Featherweight Java calculus and prove formally that all program variants generated from a well-typed PL are well-typed. We could use these works to verify in practice the well-formedness constraint used in the PL definition of our theory (see Definition 9), depending on the asset language used.

The theory we present in this article aims to formalize concepts and processes from tools [47,49,50] and practical experience [51,5,45,17,46,48] on PL refactoring. We can use the refinement theory presented for deriving more interesting PL refactoring tools.

Lotufo et al. [52] analyzed 21 versions of the Linux kernel over five years. They analyze how a number of characteristics, such as number of features, height of the tree and depth of the leaves, using the FMs of those versions. Based on this investigation, they identify challenges encountered in the process. We intend to use our catalog of refactorings to reason whether some of the transformations performed in the Linux kernel are refinements according to our definition. Initial results show that we can use the templates presented in this work to justify safe evolution of two different Java-based PLs [38]. A more rigorous evaluation of the proposed theory is, however, left as future work.

6. Conclusions

In our previous work [15,6], we propose a notion and a formal refinement theory for product lines. In this work, we revise that theory to obtain more generality and explore one of its major applications, the derivation of refinement transformation templates that we can use as a basis for constructing a product line refactoring catalogue. We first instantiate the theory with formalizations of existing specific languages for feature model and configuration knowledge. Based on that, we propose a number of refinement transformation templates that justify safe product line evolution scenarios such as adding an optional feature and splitting code assets for extracting variations. Using PVS encodings of the refinement theory and the specific languages, we prove that the proposed transformation templates are sound with respect to the refinement notion.

As the transformation templates precisely specify the transformation mechanics and preconditions, their soundness is especially useful for correctly implementing the transformations and avoiding typical problems with current single program refactoring tools [53,54]. In fact, soundness could help to avoid even subtler problems that can appear with product line refactoring tools.

We could achieve similar benefits by instantiating our theory with alternative languages for feature model and configuration knowledge. Indeed, some of the proposed templates are independent of one of the two concrete languages we use here. Therefore their proofs could be easily adapted for a different context, helping to establish product line refinement catalogues for other languages. This generality actually applies to asset languages. As we do not instantiate the theory with a particular asset language, all the templates we propose can be applied to product lines using any asset language that complies to the corresponding interface we define here.

As future work, we intend to create a comprehensive product line refinement transformation catalog and instantiate our theory with other feature model and configuration knowledge languages. We also intend to instantiate our theory with asset languages and associated refinement notions. By composing the PL templates with the asset templates, we could propose asset language specific templates. We intend to use ROOL [9] or another subset of Java [55] with a reference semantics. We also aim at evaluating this catalog by formally deriving refactorings in real case studies, such as the Linux Kernel. Finally, we intend to evolve our refactoring product line tool [50] to make it safer by taking into consideration the results presented here.

Acknowledgements

We thank the anonymous referees for the useful suggestions and comments to improve this work. We would also like to thank colleagues of the Software Productivity Group for helping to significantly improve this work. Laís Neves helped to identify some of the templates we present. 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. Finally, we would like to acknowledge current financial support from CNPq, FACEPE, and CAPES projects, and early support from FINEP and Meantime mobile creations. This work was also supported by the National Institute of Science and Technology for Software Engineering (INES[7]).

Appendix. Extra proofs

In this appendix we present proofs for theorems we omitted in the main text. Some of the proofs require additional lemmas and definitions, which we also detail in here. The PVS specification of the whole theory, and proof files for all lemmas and theorems are available at the online appendix.

Definition 12 (*Auxiliary Asset Mapping Functions*)**.** Let m be an *AssetMapping*.

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

Lemma 12 (*Asset Mapping Domain Membership*)**.** *For asset mapping A, asset name n, and asset a, if*

$$A(n) = a$$

then

$$n \in dom(A).$$

Proof. For arbitrary A, n, and a, assume $A(n) = a$. By Definition 12 (*dom*), we have to prove that

$$\exists x : Asset \mid A(n) = x.$$

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

Lemma 13 (*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

$$A\langle S\rangle = \emptyset.$$

Proof. For arbitrary A and S, assume $\neg\exists n \in S \cdot n \in dom(A)$. By Definition 12 ($A\langle\rangle$), we have to prove that

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

By Lemma 12, we then have to prove that

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

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

Lemma 14 (*Distributed Mapping Over Singleton*)**.** *For asset mapping A, asset name n, and finite set of asset names S, if*

$$n \in dom(A)$$

then

$$\exists a : Asset \cdot A(n) = a \ \wedge \ A\langle\{n\} \cup S\rangle = \{a\} \cup A\langle S\rangle.$$

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

Proof. For arbitrary A, n, and S, assume $n \in dom(A)$. From this, Definition 12 (dom), and set comprehension and membership properties, we have

$$\exists a : Asset \cdot A(n) = a. \tag{21}$$

Let a_1 be such a. By Definition 12 ($A\langle\rangle$), we have

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

By set membership and comprehension properties, we have

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

By Definition 12 ($A\langle\rangle$), our assumption that A is an AM, and set membership and comprehension properties, we have

$$A\langle\{n\} \cup S\rangle = \{a_1\} \cup A\langle S\rangle.$$

From this and remembering that (21) was instantiated with a_1, a_1 provides the a we need to conclude the proof. □

Theorem 3 (*Asset Mapping Compositionality*)**.** *For asset mapping A and A′, if*

$$A \sqsubseteq A'$$

then

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

Proof. For arbitrary A and A', assume $A \sqsubseteq A'$. By Definition 7, this amounts to

$$\begin{aligned} &dom(A) = dom(A') \\ &\wedge \forall n \in dom(A) \cdot \\ &\quad \exists a, a' : Asset \cdot A(n) = a \wedge A'(n) = a' \wedge a \sqsubseteq a'. \end{aligned} \tag{22}$$

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

$$\begin{aligned} &\forall ans' : \mathcal{F}[AssetName] \cdot \\ &\quad card(ans') < card(ans) \\ &\quad \Rightarrow \forall as : \mathcal{F}[Asset] \cdot \\ &\qquad wf(as \cup A\langle ans'\rangle) \\ &\qquad \Rightarrow wf(as \cup A'\langle ans'\rangle) \wedge as \cup A\langle ans'\rangle \sqsubseteq as \cup A'\langle ans\rangle \end{aligned} \tag{23}$$

and we have to prove

$$\begin{aligned} &\forall as : \mathcal{F}[Asset] \cdot \\ &\qquad wf(as \cup A\langle ans\rangle) \\ &\quad \Rightarrow wf(as \cup A'\langle ans\rangle) \wedge as \cup A\langle ans\rangle \sqsubseteq as \cup A'\langle ans\rangle. \end{aligned} \tag{24}$$

By case analysis, now consider that $\neg(\exists an \in ans \cdot an \in dom(A))$. By Lemma 13, we have that $A\langle ans\rangle = \emptyset$. Similarly, given that $dom(A) = dom(A')$ (see (22)), we also have that $A'\langle ans\rangle = \emptyset$. So, by set union properties, we are left to prove that

$$\forall as : \mathcal{F}[Asset] \cdot wf(as) \Rightarrow wf(as) \wedge as \sqsubseteq as.$$

The proof trivially follows from asset set refinement reflexivity (see Axiom 1) and propositional calculus.

Let us 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 (24), we are left to prove that

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

By Lemma 14, 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 (22), we also have that $a \sqsubseteq a'$. By equational reasoning, we then have to prove that

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

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

$$\begin{aligned} & wf(as \cup \{a'\} \cup A'\langle ans'\rangle) \\ \wedge & as \cup \{a\} \cup A\langle ans'\rangle \sqsubseteq as \cup \{a'\} \cup A'\langle ans'\rangle. \end{aligned} \tag{25}$$

By the induction hypothesis (see (23)), instantiating ans' with the ans' just introduced, note that we will have $card(ans') < card(ans)$ and, therefore

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

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

$$\begin{aligned} & wf(as \cup \{a\} \cup A'\langle ans'\rangle) \\ \wedge & as \cup \{a\} \cup A\langle ans'\rangle \sqsubseteq as \cup \{a\} \cup A'\langle ans\rangle. \end{aligned}$$

Now, given that $a \sqsubseteq a'$, from the compositionality axiom (Axiom 2) and the above we have that

$$\begin{aligned} & wf(as \cup \{a'\} \cup A'\langle ans'\rangle) \\ \wedge & as \cup \{a\} \cup A\langle ans'\rangle \sqsubseteq as \cup \{a'\} \cup A'\langle ans'\rangle. \end{aligned}$$

The proof then follows from (25), the above, and Axiom 1. □

Theorem 5 (*Instantiation of Axiom 3*). *For asset mapping A and A', if*

$$A \sqsubseteq A'$$

then

$$\begin{aligned} & \forall K : CK, c : Configuration \cdot \\ & \quad wf([\![K]\!]_c^A) \\ & \Rightarrow wf([\![K]\!]_c^{A'}) \wedge [\![K]\!]_c^A \sqsubseteq [\![K]\!]_c^{A'}. \end{aligned}$$

Proof. For arbitrary A and A', assume $A \sqsubseteq A'$. For arbitrary K and c, we have to prove

$$\begin{aligned} & wf([\![K]\!]_c^A) \\ \Rightarrow & wf([\![K]\!]_c^{A'}) \wedge [\![K]\!]_c^A \sqsubseteq [\![K]\!]_c^{A'}. \end{aligned}$$

By Definition 5, which defines the *semantics* function used to instantiate the general theory, this amounts to

$$\begin{aligned} & wf(A\langle eval(K,c)\rangle) \\ \Rightarrow & wf(A'\langle eval(K,c)\rangle) \wedge A\langle eval(K,c)\rangle \sqsubseteq A'\langle eval(K,c)\rangle. \end{aligned} \tag{26}$$

From Theorem 3, instantiated with A and A', we have

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

Instantiating *ans* with $eval(K,c)$ and *as* with $\emptyset$, we have

$$\begin{aligned} & wf(\emptyset \cup A\langle eval(K,c)\rangle) \\ \Rightarrow & wf(\emptyset \cup A'\langle eval(K,c)\rangle) \wedge \emptyset \cup A\langle eval(K,c)\rangle \sqsubseteq \emptyset \cup A'\langle eval(K,c)\rangle. \end{aligned}$$

By set properties, we have

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

which is what we have to prove—see (26). □

References

[1] K. Pohl, G. Böckle, F. van der Linden, Software Product Line Engineering: Foundations, Principles and Techniques, Springer, 2005.
[2] F. van der Linden, K. Schmid, E. Rommes, Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering, Springer, 2007.
[3] C. Krueger, Easing the transition to software mass customization, in: 4th International Workshop on Software Product-Family Engineering, in: LNCS, vol. 2290, Springer, 2002, pp. 282–293.
[4] P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley, 2001.
[5] V. Alves, P. Matos Jr., L. Cole, P. Borba, G. Ramalho, Extracting and evolving mobile games product lines, in: 9th International Software Product Line Conference, in: LNCS, vol. 3714, Springer, 2005, pp. 70–81.

[6] P. Borba, An introduction to software product line refactoring, in: Generative and Transformational Techniques in Software Engineering III, in: LNCS, vol. 6491, Springer, 2011, pp. 1–26.
[7] W. Opdyke, Refactoring object-oriented frameworks, Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1992.
[8] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.
[9] P. Borba, A. Sampaio, A. Cavalcanti, M. Cornélio, Algebraic reasoning for object-oriented programming, Science of Computer Programming 52 (2004) 53–100.
[10] L. Cole, P. Borba, Deriving refactorings for AspectJ, in: 4th International Conference on Aspect-Oriented Software Development, ACM, 2005, pp. 123–134.
[11] K. Kang, S. Cohen, J. Hess, W. Novak, A.S. Peterson, Feature-oriented domain analysis (FODA) feasibility study, Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.
[12] K. Czarnecki, U. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.
[13] C.A.R. Hoare, Proof of correctness of data representations, Acta Informatica (1972) 271–281.
[14] E. Dijkstra, Notes on Structured Programming, Academic Press, 1971.
[15] P. Borba, L. Teixeira, R. Gheyi, A theory of software product line refinement, in: 7th International Colloquium Conference on Theoretical Aspects of Computing, Springer, 2010, pp. 15–43.
[16] S. Owre, J. Rushby, N. Shankar, PVS: a prototype verification system, in: 11th International Conference on Automated Deduction, Springer, 1992, pp. 748–752.
[17] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, C. Lucena, Refactoring product lines, in: 5th International Conference on Generative Programming and Component Engineering, 2006, pp. 201–210.
[18] R. Gheyi, T. Massoni, P. Borba, Algebraic laws for feature models, Journal of Universal Computer Science 14 (21) (2008) 3573–3591.
[19] K. Czarnecki, S. Helsen, U. Eisenecker, Formalizing cardinality-based feature models and their specialization, Software Process: Improvement and Practice 10 (1) (2005) 7–29.
[20] D. Batory, Feature models, grammars, and propositional formulas, in: 9th International Software Product Lines Conference, in: LNCS, vol. 3714, Springer, 2005, pp. 7–20.
[21] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, Y. Bontemps, Generic semantics of feature diagrams, Computer Networks 51 (2) (2007) 456–479.
[22] R. Bonifácio, P. Borba, Modeling scenario variability as crosscutting mechanisms, in: 8th International Conference on Aspect-Oriented Software Development, 2009, pp. 125–136.
[23] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Filho, F. Dantas, Evolving software product lines with aspects: an empirical study on design stability, in: 30th International Conference on Software Engineering, 2008, pp. 261–270.
[24] D.M. Weiss, J.J. Li, J.H. Slye, T.T. Dinh-Trong, H. Sun, Decision-model-based code generation for SPLE, in: 12th International Software Product Line Conference, 2008, pp. 129–138.
[25] A. Sampaio, P. Borba, Transformation laws for sequential object-oriented programming, in: Refinement Techniques in Software Engineering, in: LNCS, vol. 3167, Springer, 2004, pp. 18–63.
[26] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, Getting started with AspectJ, Communications of the ACM 44 (10) (2001) 59–65.
[27] K. Czarnecki, K. Pietroszek, Verifying feature-based model templates against well-formedness OCL constraints, in: 5th International Conference on Generative Programming and Component Engineering, 2006, pp. 211–220.
[28] C. Gacek, M. Anastasopoulos, Implementing product line variabilities, SIGSOFT Software Engineering Notes 26 (3) (2001) 109–117.
[29] Spivey, The Z Notation: A Reference Manual, Prentice Hall, 1987.
[30] S. Zschaler, P. Sánchez, J. Santos, M. Alférez, A. Rashid, L. Fuentes, A. Moreira, J. Araújo, U. Kulesza, VML*—a family of languages for variability management in software product lines, in: 2nd International Conference on Software Language Engineering, 2009, pp. 82–102.
[31] D.S. Batory, Feature-oriented programming and the AHEAD tool suite, in: 26th International Conference on Software Engineering, 2004, pp. 702–703.
[32] S. Apel, C. Kästner, C. Lengauer, Feature Featherweight Java: a calculus for feature-oriented programming and stepwise refinement, in: 7th International Conference on Generative Programming and Component Engineering, 2008, pp. 101–112.
[33] R. Gheyi, T. Massoni, P. Borba, An abstract equivalence notion for object models, Electronic Notes in Theoretical Computer Science 130 (2005) 3–21.
[34] T. Massoni, R. Gheyi, P. Borba, Formal model-driven program refactoring, in: 11th International Conference on Fundamental Approaches to Software Engineering, in: LNCS, vol. 4961, Springer, 2008, pp. 362–376.
[35] S. Thaker, D. Batory, D. Kitchin, W. Cook, Safe composition of product lines, in: 6th International Conference on Generative Programming and Component Engineering, 2007, pp. 95–104.
[36] C. Kästner, S. Apel, T. Thüm, G. Saake, Type checking annotation-based product lines, ACM Transactions on Software Engineering and Methodology (in press).
[37] J.A. Goguen, J. Meseguer, Security policies and security models, in: IEEE Symposium on Security and Privacy, 1982, p. 11.
[38] L. Neves, L. Teixeira, D. Sena, V. Alves, U. Kulesza, P. Borba, Investigating the safe evolution of software product lines, in: 10th International Conference on Generative Programming and Component Engineering, 2011, pp. 33–42.
[39] W. Opdyke, R. Johnson, Refactoring: An aid in designing application frameworks and evolving object-oriented systems, in: Symposium on Object-Oriented Programming emphasizing Practical Applications, 1990, pp. 145–160.
[40] C. Morgan, Programming from Specifications, second ed., Prentice Hall, 1994.
[41] A. Roscoe, C. Hoare, The laws of occam programming, Theoretical Computer Science 60 (2) (1988) 177–229.
[42] C.A.R. Hoare, J. Spivey, I. Hayes, J. He, C. Morgan, A. Roscoe, J. Sanders, I. Sorenson, B. Sufrin, Laws of programming, Communications of the ACM 30 (8) (1987) 672–686.
[43] L. Silva, A. Sampaio, Z. Liu, Laws of object-orientation with reference semantics, in: 6th IEEE International Conference on Software Engineering and Formal Methods, 2008, pp. 217–226.
[44] M. Critchlow, K. Dodd, J. Chou, A. van der Hoek, Refactoring product line architectures, in: 1st International Workshop on Refactoring: Achievements, Challenges, and Effects, 2003, pp. 23–26.
[45] R. Kolb, D. Muthig, T. Patzke, K. Yamauchi, A case study in refactoring a legacy component for reuse in a product line, in: 21st International Conference on Software Maintenance, 2005, pp. 369–378.
[46] S. Trujillo, D. Batory, O. Diaz, Feature refactoring a multi-representation program into a product line, in: 5th International Conference on Generative Programming and Component Engineering, 2006, pp. 191–200.
[47] J. Liu, D. Batory, C. Lengauer, Feature oriented refactoring of legacy applications, in: 28th International Conference on Software Engineering, 2006, pp. 112–121.
[48] C. Kastner, S. Apel, D. Batory, A case study implementing features using AspectJ, in: 11th International Software Product Line Conference, 2007, pp. 223–232.
[49] F. Calheiros, P. Borba, S. Soares, V. Nepomuceno, V. Alves, Product line variability refactoring tool, in: 1st Workshop on Refactoring Tools, 2007, pp. 33–34.
[50] V. Alves, F. Calheiros, V. Nepomuceno, A. Menezes, S. Soares, P. Borba, FLiP: Managing software product line extraction and reaction with aspects, in: 12th International Software Product Line Conference, 2008, p. 354.
[51] V. Alves, I. Cardim, H. Vital, P. Sampaio, A. Damasceno, P. Borba, G. Ramalho, Comparative analysis of porting strategies in J2ME games, in: 21st IEEE International Conference on Software Maintenance, 2005, pp. 123–132.
[52] R. Lotufo, S. She, T. Berger, K. Czarnecki, A. Wasowski, Evolution of the Linux kernel variability model, in: 14th International Conference on Software Product Lines, 2010, pp. 136–150.
[53] F. Steimann, A. Thies, From public to private to absent: Refactoring Java programs under constrained accessibility, in: 23rd European Conference on Object-Oriented Programming, in: LNCS, vol. 5653, Springer, 2009, pp. 419–443.
[54] G. Soares, R. Gheyi, D. Serey, T. Massoni, Making program refactoring safer, IEEE Software 27 (2010) 52–57.
[55] A. Banerjee, D.A. Naumann, Secure information flow and pointer confinement in a Java-like language, in: 15th IEEE workshop on Computer Security Foundations, 2002, pp. 253–267.