An Introduction to Software Product Line Refactoring

Paulo Borba

Informatics Center
Federal University of Pernambuco
phmb@cin.ufpe.br

Abstract. Although software product lines (PLs) can bring significant productivity and quality improvements through strategic reuse, bootstrapping existing products into a PL, and extending a PL with more products, is often risky and expensive. These kinds of PL derivation and evolution might require substantial effort and can easily affect the behavior of existing products. To reduce these problems, we propose a notion of product line refactoring and associated transformation templates that should be part of a PL refactoring catalogue. We discuss how the notion guides and improves safety of the PL derivation and evolution processes; the transformation templates, particularly when automated, reduce the effort needed to perform these processes.

1 Introduction

A software product line (PL) is a set of related software products that are generated from reusable assets. Products are related in the sense that they share common functionality. Assets correspond to components, classes, property files, and other artifacts that are composed in different ways to specify or build the different products. For example, in the simplified version of the Rain of Fire mobile game product line shown in Fig. 1, we have three products varying only in how they support clouds in the game background, as this impacts on product size and therefore demands specific implementations conforming to different mobile phones' memory resources. The classes and images are reused by the three products; the clouds image, for instance, is used by two products. Each XML file and aspect [KHH+01] (`.aj` file), which are common variability implementation mechanisms [GA01, AJC+05], specify cloud specific data and behavior, so are only reused when considering other products not illustrated in Fig. 1.

This kind of reuse targeted at a specific set of products can bring significant productivity and time to market improvements [PBvdL05, Chapter 21][vdLSR07, Chapters 9-16]. The extension of a PL with a new product demands less effort because existing assets can likely make up to a large part of the new product. The maintenance of existing products also demands less effort because changes in a single asset often have an impact on more than one product. Indirectly, we can improve quality too [vdLSR07], since assets are typically more exposed and tested through their use in different products.

R. Lämmel et al. (Eds.): GTTSE 2009, LNCS 6491, pp. 1–26, 2011.

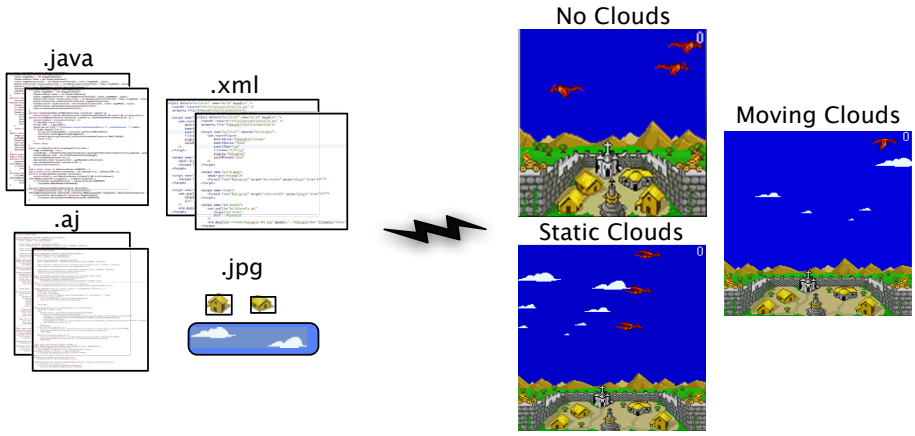

Fig. 1. Mobile game product line

To obtain these benefits with reduced upfront investment, previous work proposes to minimize the initial product line analysis and development process by bootstraping existing related products into a PL [Kru02, CN01, AJC+05]. Through an extraction process, we can separate variations from common parts, and then discard duplicate common parts. In Fig. 2, the first pair of arrows and the subsequent arrow illustrate this process for the code[1] of two related products: the same mobile game with moving clouds, differing only in their image loading policy. A similar process applies for evolving a PL, when adding new products might require extracting a part shared by existing products but not appropriate for some new products. In this case, a previously common part becomes a variation in the new PL. For example, the rightmost arrow in Fig. 2 shows that we extracted the code associated to clouds to a new aspect, which is an adequate mechanism to separate the cloud variations from the other parts of the game. We can now, by not including the Clouds aspect, have games without clouds in the product line.

Although useful to reduce upfront investment and the risk of losing it due to project failure, this extraction process might be tedious. Manually extracting and changing different parts of the code requires substantial effort, especially for analyzing the conditions that ensure the correctness of the extraction process. In fact, this process can easily introduce defects, modifying the behavior exhibited by products before the extraction process, and compromising the promised benefits on other dimensions of costs and risks.

To minimize these problems, the proposed extraction process could benefit from automatic refactorings: behavior-preserving source-to-source transformations that improve some quality factor, usually reusability and maintainability [Rob99, Fow99]. Refactorings automate tedious changes and analyses,

[1] The same process applies for other artifacts such as requirements documents, test cases, and design models [TBD06].

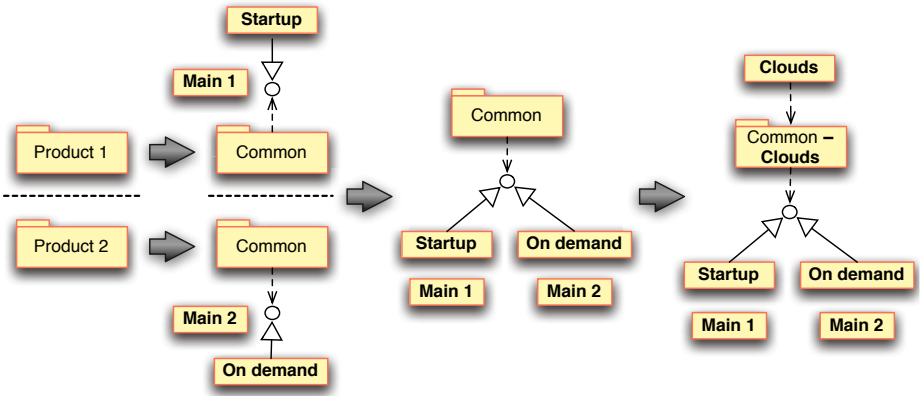

Fig. 2. Product line extraction

consequently reducing costs and risks.[2] They can help too by providing guidance on how to structure extracted variants. All that could be useful for deriving a PL from existing products, and also for evolving a PL by simply improving its design or by adding new products while preserving existing ones.

However, existing refactoring notions [Opd92, Fow99, BSCC04, CB05] focus on transforming single programs, not product lines (PLs). These notions, therefore, might justify the pair of transformations in the first step of Fig. 2, for independently transforming two programs [TBD06, KAB07, AJC+05, AGM+06]. But they are not able to justify the other two steps. The second requires merging two programs into a product line; the resulting product line has conflicting assets such as the two main classes, so it does not actually correspond to a valid program. Similarly, the third step involves transforming conflicting assets, which again are not valid programs.

Similar limitations apply for refactoring of other artifacts, including design models [GMB05, MGB08]. These notions focus on refactoring a single product, not a product line. Besides that, as explained later, in a product line we typically need extra artifacts, such as feature models [KCH+90, CE00], for automatically generating products from assets. So a PL refactoring notion should overcome the single product limitations just mentioned and go beyond reusable assets, transforming the extra artifacts as well.

We organize this text as follows. Section 2 introduces basic concepts and notation for feature models and other extra product line artifacts [CE00, BB09]. This section also emphasizes informal definitions for the semantics of these artifacts, and for a notion of program refinement [SB04, BSCC04], with the aim of explicitly introducing notation and the associated intuitions. This aligns with our focus on establishing concepts about an emerging topic, rather than providing a complete formalization or reporting practical experience on product line

[2] Current refactoring tools might still introduce defects because they do not fully implement some analyses [ST09].

refactoring [ACV+05, AJC+05, AGM+06, TBD06, CBS+07, KAB07, ACN+08]. Following that, in Sec. 3, we propose and formalize a notion of product line refactoring. This goes beyond refactoring of feature models, which is the focus of our previous work [AGM+06, GMB08]. The notion and the formalization are independent of languages used, for example, to describe feature models and code assets. They depend only on the interfaces expressed by the informal definitions that appear in Sec. 2; that is why the theorems in Sec. 3 do not rely, for example, on a formalization of feature models. To illustrate one of the main applications of such a refactoring notion, Sec. 4 shows different kinds of transformation templates that can be useful for refactoring product lines. This goes beyond templates for transforming feature models [AGM+06, GMB08], considering also other artifacts, both in isolation and in an integrated way. As the templates use specific notation for feature models and the other artifacts, proving that the transformations are sound with respect to the refactoring notion requires the formal semantics of the used notations. However, as our main aim is to stimulate the derivation of comprehensive refactoring catalogues, considering different notations and semantic formalizations for product line artifacts [CHE05, Bat05, SHTB07, GMB08], we prefer not to lose focus by introducing details of the specific notations adopted here.

2 Software Product Line Concepts

In the PL approach adopted in this text, Feature Models (FMs) and Configuration Knowledge (CK) [CE00] enable the automatic generation of products from assets. 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 CK relates features and assets, specifying which assets implement possible feature combinations. Hence we use a CK to actually build a product given chosen features for that product. We now explain in more detail these two kinds of artifacts, using examples from the Mobile Media product line [FCS+08], which contains applications – such as the one illustrated in Fig. 3 – that manipulate photos, music, and video on mobile devices.

2.1 Feature Models

A feature model is essentially represented as a tree, containing features and information about how they are related. Features basically abstract groups of associated requirements, both functional and non-functional. 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. 4 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.

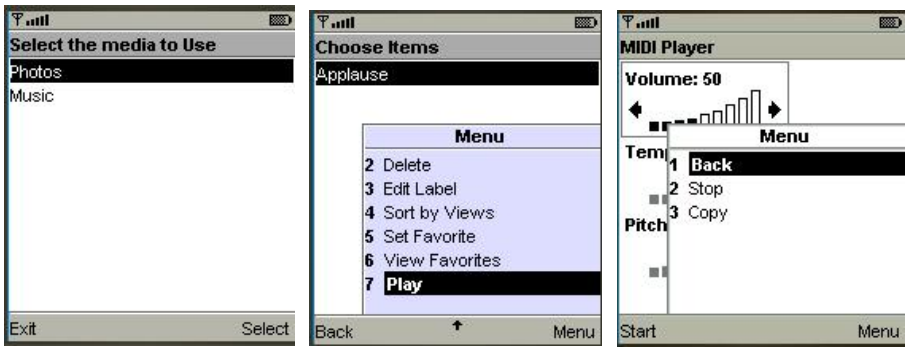

Fig. 3. Mobile Media screenshots

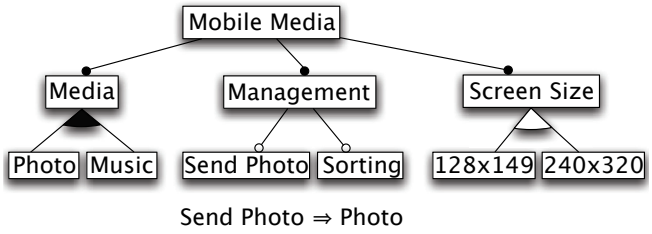

Fig. 4. Mobile Media simplified feature model

Besides these relationships, feature models may contain propositional logic formulas about features. We use feature names as atoms to indicate that a feature should be selected. So negation of a feature indicates that it should not be selected. For instance, the formula just below the tree in Fig. 4 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 possible configuration, but {Music, Photo, 240x320, 128x149} is not because it breaks the Screen Size alternative constraint. In summary, a valid configuration is one that satisfies all FM constraints, specified both graphically and through formulas. Each valid configuration corresponds to one PL product, expressed in terms of the features it supports. So the following definition captures the intuition that a FM denotes the set of products in a PL.

Definition 1 ⟨FM semantics⟩
The semantics of a feature model F, represented as $[\![F]\!]$, is the set of all valid product configurations (sets of feature names) of F.

□

To introduce the notion of PL refactoring, discuss refactorings of specific PLs, and even derive general theorems about this notion, this is all we need to know

about FMs. So we omit here the formal definitions of $[\![F]\!]$ and valid configurations, which appear elsewhere [GMB08, AGM+06] for the notation used in this section. There are alternative FM notations [CHE05, Bat05, SHTB07, GMB08], but, as shall be clear later, our notion of PL refactoring does not depend on the used FM notation. Our notion works for any FM notation whose semantics can be expressed as a set of configurations, as explicitly captured by Definition 1. On the other hand, to prove soundness of refactoring transformation templates, as showed later, we would need a formal definition of $[\![F]\!]$ because the templates use a specific FM notation. But this is out of the scope of this work, which aims to establish PL refactoring concepts and stimulate further work – such as the derivation of refactoring catalogues considering different notations and semantic formalizations for PL artifacts – rather than to develop a theory restricted to a specific FM notation.

2.2 Configuration Knowledge

As discussed in the previous section, features are groups of requirements, so they must be related to the assets that realize them. Abstracting some details of previous work [BB09], a CK is a relation from feature expressions (propositional formulas having feature names as atoms) to sets of asset names. For example, showing the relation in tabular form, the following CK

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

establishes that if the Photo and Music features are both selected then the `AppMenu` aspect, 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 [GA01, AJC+05] that has the effect of presenting the left screenshot in Fig. 3.[3] 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, the evaluation of a CK yields the names of the assets needed to build the corresponding product. In our example, the configuration {Photo, 240x320}[4] leads to

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

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

[4] Remember we omit mandatory features for brevity.

This gives the basic intuition for defining the semantics of a CK.

Definition 2 ⟨CK semantics⟩
The semantics of a configuration knowledge K, represented as $[\![K]\!]$, is a function that maps product configurations into sets of asset names, in such a way that, for a configuration c, an asset name a is in the set $[\![K]\!]c$ iff there is a row in K that contains a and its expression evaluates to true according to c. □

We again omit the full formalization since, as discussed before, it is only necessary to formally prove soundness of PL refactoring transformation templates that use this particular notation for specifying the CK. Our notion of PL refactoring, and associated properties, work for any CK notation whose semantics can be expressed as a function that maps configurations into sets of assets names. This is why we emphasize the $[\![K]\!]$ notation in Definition 2, and use it later when defining PL refactoring.

2.3 Assets

Although the CK in the previous section refers only to code assets, in general we could also refer to requirements documents, design models, test cases, image files, XML files, and so on. For simplicity, here we focus on code assets as they are equivalent to other kinds of assets with respect to our interest in PL refactoring. The important issue here is not the nature of the asset contents, but how we compare assets and refer to them in the CK. We first discuss the asset reference issue.

Asset mapping. With respect to CK references, remember that we might have conflicting assets in a PL. For instance, on the right end of Fig. 2, we have two `Main` classes. They have the same name and differ only on how they instantiate their image loading policy:

```
class Main {                          class Main {
    ...new StartUp(...);...               ...new OnDemand(...);...
}                                     }
```

We could avoid the duplicate class names by having a single `Main` class that reads a configuration file and then decides which policy to use, but we would then need two configuration files with the same name. Duplicate names might also apply to other classes and assets in a PL, so references to those names would also be ambiguous.

To avoid the problem, we assume that the names that appear in a CK might not exactly correspond to the names used in asset declarations. For instance, the names "Main 1" and "On demand" in this CK

⋮	⋮
On Demand	Main 2, On demand
Start Up	Main 1, Startup
On Demand ∨ Start Up	Common.java
⋮	⋮

are not really class names. Instead, the PL keeps a mapping such as the one in Fig. 5, from the CK names to actual assets. So, besides a FM and a CK, a PL actually contains an asset mapping, which basically corresponds to an environment of asset declarations.

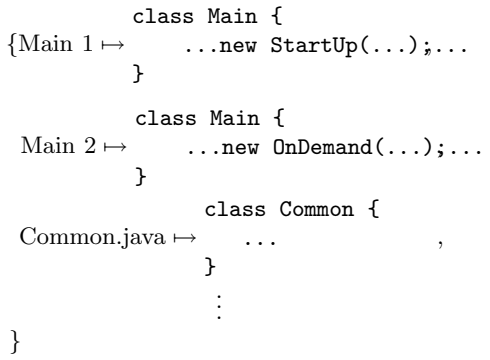

Fig. 5. Asset mapping

Asset refinement. Finally, for defining PL refactoring, we must introduce a means of comparing assets with respect to behavior preservation. As we focus on code, we use existing refinement definitions for sequential programs [SB04, CB05].

Definition 3 ⟨Program refinement⟩
For programs p_1 and p_2, p_1 is refined by p_2, 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 satisfy every input-output specification satisfied by p_1. We say that p_2 preserves the (observable) behavior of p_1.

□

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. So, 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 4 ⟨Class declaration refinement⟩
For sets of class declarations cds_1 and cds_2, cds_1 is refined by cds_2, 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 [SB04].

This definition captures the notion of behavior preservation for classes, so, using an abstract programming notation [BSCC04, SB04], code transformations of typical object-oriented refactorings can be expressed as refinements. For example, the following equivalence (refinement in both directions) establishes that we can move a public attribute a from a class C to a superclass B, and vice-versa.

class B **extends** A
  ads
  ops
end
class C **extends** B
  pub $a : T; ads'$
  ops'
end
	$=_{cds,m}$	**class** B **extends** A
		  pub $a : T; ads$
		  ops
		end
		class C **extends** B
		  ads'
		  ops'
		end

To move the attribute up to B, it is required that this does not generate a name conflict: no subclass of B, other than C, can declare an attribute with the same name, to avoid attribute redefinition or hiding. We can move a from B to C provided that a is used only as if it were declared in C. The formal pre-conditions, and a comprehensive set of similar transformations, appear elsewhere [BSCC04, SB04]. We omit them here because they focus on a specific programming language, whereas we want to establish a notion of PL refactoring that is language independent, as long as the language has a notion of program refinement such as the one just discussed. Even the overall PL refactoring templates that we show later depend on specific FM and CK languages but are programming language independent.

We could also have a similar definition of refinement for aspects and other code artifacts, but as they can all be expressed in terms of program refinement, hereafter we use the fundamental program refinement relation $\sqsubseteq$, and assume translations, for example of a set of classes with a specific main class, into the formats discussed in this section. In summary, such a reflexive and transitive notion of program refinement is all we need for our notion of PL refactoring, its basic properties, and the overall PL templates we discuss later. For reasoning about reusable assets in isolation we also need a notion of asset refinement

(such as the one for class declaration). This should be compositional, in the sense that refining an asset that is part of a given program implies refinement of the whole program.

2.4 Product Lines

We can now provide a precise definition for product lines, and a better account than what we represent in Fig. 1, which illustrates only assets and products. In particular, we make sure the three PL elements discussed in the previous sections are consistent in the sense of referring only to themselves. We also require each PL product to be a valid program[5] in its target languages.

Definition 5 ⟨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 the expressions in K refer only to features in F, the asset names in K refer only to the domain of A, and, for all $c \in [\![F]\!]$,

$$A\langle [\![K]\!]c \rangle$$

is a valid program, where $A\langle S \rangle$, for a mapping A and a set S, is an abbreviation for $\{A(s) \mid s \in S\}$.

□

For object-oriented languages, $A\langle [\![K]\!]c \rangle$ should be a well-typed set of classes containing a single main class, with a main method. This validity constraint in the definition is necessary because missing an entry on a CK might lead to products that are missing some parts and are thus invalid. Similarly, a mistake when writing a CK or asset mapping entry might yield an invalid program due to conflicting assets, like two aspects that we use as variability mechanism and introduce methods with the same signature in the same class. Here we demand CKs to be correct as explained.

It is useful to note that, following this definition, we can see a single system as a single-product PL:

The feature model contains a single feature Root and no constraints, the asset mapping A simply maps declared names to the corresponding system asset declarations, and the CK contains a single entry relating Root to all asset names. For a product line in this form, the corresponding single system is simply the image of A.

[5] Remember our focus on code artifacts. In general, we should require products to contain valid artifacts no matter the languages and notations (modeling, scripting, programming, etc.) used to describe them.

3 Product Line Refactoring

Now that we better understand what a PL is, we can define a notion of PL refactoring to address the issues mentioned in Sec. 1. Similar to program refactorings, PL refactorings are behavior-preserving transformations that improve some quality factor. However, they go beyond source code, and might transform FMs and CKs as well. We illustrate this in Fig. 6, where we refactor the simplified Mobile Media product line by renaming the feature Music. As indicated by check marks, this renaming requires changing the FM, CK, and asset mapping; due to a class name change, we must apply a global renaming, so the main method and other classes beyond `Music.java` chang too.

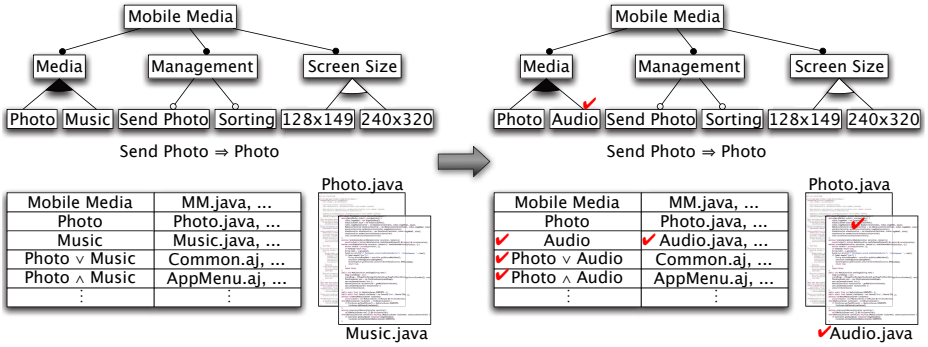

Fig. 6. PL renaming refactoring

The notion of behavior preservation should be also lifted from programs to product lines. In a PL refactoring, the resulting PL should be able to generate products (programs) 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 refactoring, 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 be always like that. We consider that the better product line might generate more products than the original one. As long as it generates enough products to match the original PL, users have no reason to complain. For instance, by adding the optional Copy feature (see Fig. 7), we refactor our example PL. The new PL generates twice as many products as the original one, but 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.

3.1 Formalization

We formalize these ideas as a notion of refinement for product lines, defined in terms of program refinement (see Definition 3). Each program generated by the original PL must be refined by some program of the new, improved, PL.

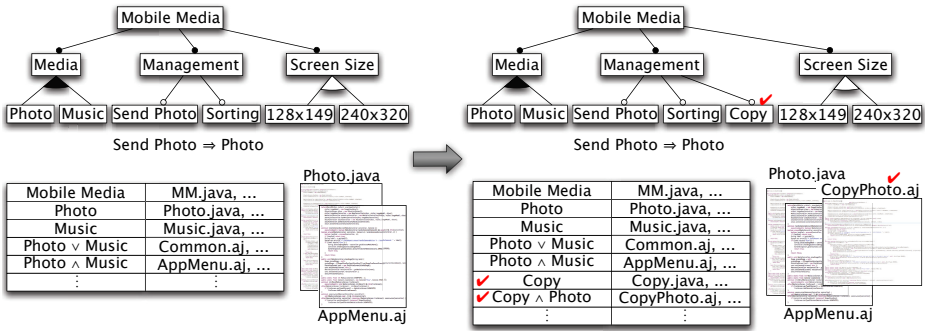

Fig. 7. Adding an optional feature refactoring

Definition 6 ⟨PL refinement⟩
For product lines (F, A, K) and (F', A', K'), the first is refined by the second, denoted

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

whenever

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

□

Remember that, for a configuration c, configuration knowledge K, and asset mapping A related to a given PL, $A\langle [\![K]\!] c\rangle$ is a set of assets that constitutes a valid program. So $A\langle [\![K]\!] c\rangle \sqsubseteq A'\langle [\![K']\!] c'\rangle$ refers to the program refinement notion discussed in Sec. 2.3.

Now, for defining PL refactoring, we need to capture quality improvement as well.

Definition 7 ⟨PL refactoring⟩
For product lines PL and PL', the first is refactored by the second, denoted

$$PL \lll PL'$$

whenever

$$PL \sqsubseteq PL'$$

and PL' is better than PL with respect to quality factors such as reusability and maintainability [Rob99, Fow99].

□

We provide no formalization of quality improvement, as this is subjective and context dependent. Nevertheless, this definition formalizes to a good extent, and slightly generalizes, our previous definition [AGM+06]: "a PL refactoring is a change made to the structure of a PL in order to improve (maintain or increase) its configurability, make it easier to understand, and cheaper to modify without changing the observable behavior of its original products". The difficulty of formally capturing quality improvement is what motivates the separate notions of refactoring and refinement. We can only be formal about the latter.

3.2 Examples and Considerations

To explore the definitions just introduced, let us analyze concrete PL transformation scenarios.

Feature names do not matter. First let us see how the definitions apply to the transformation depicted by Fig. 6. 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 yield

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

and

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

The resulting sets of asset names differ at most by a single element: `Audio.java` replacing `Music.java`. Finally, when applying these sets of names to both asset mappings, we obtain the same assets modulo global renaming, which is a well known refactoring for closed programs. This implies behavior-preservation and therefore program refinement, which is precisely what, by Definition 6, we need for assuring that the source PL is refined by the target PL. Refactoring, by Definition 7, follows from the fact that Audio is a more representative name for what is actually manipulated by the applications.

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

Safety for existing users only. To further explore the definitions, let us consider now the transformation shown in Fig. 7. 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 configurations common to both FMs – configurations without Copy. As the target CK is an extension of the source CK for dealing with cases when Copy is selected, evaluating the target CK with any configuration without Copy yields the same asset names yielded by the source CK with the same configuration. In this restricted name domain, both asset mappings are equal, since the target mapping is an extension of the first for names such as `CopyPhoto.java`, which appears only when we select Copy. Therefore, the resulting assets produced by each PL are the same, trivially implying program refinement and then PL

refinement. Refactoring follows because the new PL offers more reuse opportunities due to new classes and aspects such as `CopyPhoto.java`.

By focusing on the common configurations to both FMs, we check nothing about the new products offered by the new PL. In fact, they might even not operate at all. Our refactoring notion assures only that users of existing products will not be 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 refactorings are safe transformations only in the sense that we can change a PL without impacting existing users.

Non refactorings. As discussed, the transformation depicted in Fig. 6 is a refactoring. We transform classes and aspects through a global renaming, which preserves behavior for closed programs. But suppose that, besides renaming, we change the `AppMenu.aj`[6] aspect so that, instead of the menu on the left screenshot in Fig. 3, 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 and refactoring, consequently.

Despite not being a refinement, this menu change is an 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 and refactoring. They will have to test the PL to make sure the effect of the applied transformations actually corresponds to the expected behavior changes. The benefits of checking for refactoring only apply when the intention of the transformation is to improve PL configurability or internal structure, without changing observable behavior.

Similarly, adding a mandatory feature such as Logging to a PL is not a refactoring. This new feature might be an important improvement for the PL, but it deliberately changes the behavior of all products, so we cannot rely on refactoring to assure that the transformation is safe. In fact, by checking for refactoring, we learn that the transformation is not safe, as expected. If the intention was to preserve behavior, a failed refactoring check would indicate a problem in the transformation process.

Figure 8 shows another possibly useful transformation that is not a refactoring. Suppose that, for market reasons, we removed the 128x149 screen size alternative feature, so the target PL generates only half of the products generated by the source PL. Configurations such as {Music, 240x320}, without the removed screen size, yield exactly the same product in both PLs. Behavior is preserved for these configurations because the asset mappings and CKs differ only on entries related to `SS1.aj`. However, refactoring does not hold because of configurations such as {Music, 128x149}. The products associated with these configurations cannot be matched by products from the target PL, which is required for PL refinement.

[6] See Sec. 2.2 for understanding the role this aspect plays.

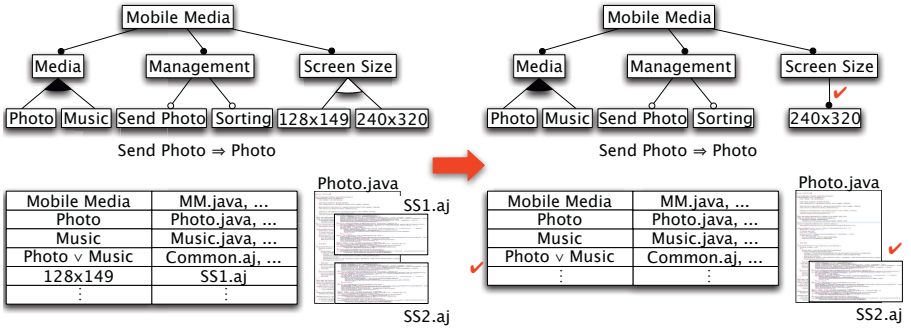

Fig. 8. Non refactoring: removing an alternative feature

3.3 Special Cases

For checking refinement as in the previous section, it is useful to know PL refinement properties that hold for special cases. First, when PLs differ only by their CK, we can check refinement by checking if the CKs differ only syntactically.

Theorem 1 ⟨Refinement with different CKs⟩
For product lines (F, A, K) and (F, A, K'), if

$$[\![K]\!] = [\![K']\!]$$

then

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

Proof: First assume that $[\![K]\!] = [\![K']\!]$. By Definition 6, we have to prove that

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

From our assumption, this is equivalent to

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

For an arbitrary $c \in [\![F]\!]$, just let c' be c and the proof follows from program refinement reflexivity.

□

Note that the reverse does not hold because the asset names generated by K and K' might differ for assets that have no impact on product behavior,[7] or for assets that have equivalent behavior but have different names in the PLs.

We can also simplify checking when only asset mappings are equal. In this case we still bypass program refinement checking, which is often difficult.

[7] Obviously an anomaly, but still possible.

Theorem 2 ⟨Refinement with equal asset mappings⟩
For product lines (F, A, K) and (F', A, K'), if

$$\forall c \in [\![F]\!] \cdot \exists c' \in [\![F']\!] \cdot [\![K]\!]c = [\![K']\!]c'$$

then

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

Proof: First assume that $\forall c \in [\![F]\!] \cdot \exists c' \in [\![F']\!] \cdot [\![K]\!]c = [\![K']\!]c'$. By Definition 6, we have to prove that

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

For an arbitrary $c \in [\![F]\!]$, our assumption gives us a $c'_1 \in [\![F']\!]$ such that

$$[\![K]\!]c = [\![K']\!]c'_1$$

Just let c' be c'_1 and the proof follows from this equality, equational reasoning, and program refinement reflexivity.

□

Again, the reverse does not hold, for similar reasons. Noting that, in the first theorem, '$[\![K]\!] = [\![K']\!]$' actually amounts to '$\forall c \in [\![F]\!] \cdot [\![K]\!]c = [\![K']\!]c$' helps to explore the similarities between the two special cases.

3.4 Population Refactoring

The PL refactoring notion discussed so far is useful to check safety when transforming a PL into another. We can then use it to justify the rightmost transformation in Fig. 2, whereas typical program refactorings justify the leftmost pair of transformations. The middle transformation, however, we cannot justify by either refactoring notion since we actually transform two programs, which we can see as two single-product PLs,[8] into a PL. To capture this merging situation, we introduce a refactoring notion that deals with more than one source PL. We have to guarantee that the target PL generates enough products to match the source PLs products. As before, we first define refinement. Here is the definition considering two source PLs.

Definition 8 ⟨Population refinement⟩
For product lines PL_1, PL_2, and PL, we say that the first two PLs are refined by the third, denoted by

$$PL_1\, PL_2 \sqsubseteq PL$$

whenever

$$PL_1 \sqsubseteq PL \wedge PL_2 \sqsubseteq PL$$

□

[8] See Sec. 2.4.

It is easy to generalize the definition for any number of source PLs, but we omit the details for simplicity. So, whereas we will likely use Definition 6 to indirectly relate product families (products with many commonalities and few differences), we will likely use Definition 8 to indirectly relate product populations (products with many commonalities but also with many differences) [vO02]. This assumes that, as a population has less commonality, it might have been initially structured as a number of PLs. In this case, the population corresponds to the union of the families generated by each source PL, or by the target PL.

Now we define population refactoring in a similar way to PL refactoring.

Definition 9 ⟨Population refactoring⟩
For product lines PL_1, PL_2, and PL, the first two are refactored by the third, denoted

$$PL_1\, PL_2 \lll PL$$

whenever

$$PL_1\, PL_2 \sqsubseteq PL$$

and PL is better than PL_1 and PL_2 with respect to quality factors such as reusability and maintainability.

□

Assuming proper FMs and CKs, population refinement justifies the middle transformation in Fig. 2. Refactoring follows from the fact that we eliminate duplicated code.

4 Product Line Refactoring Catalogue

With the refactoring notions introduced so far, we are able to derive a PL from existing products, and also to evolve a PL by simply improving its design or by adding new products while preserving existing ones. In this way we essentially handle the problems mentioned in Sec. 1. Nevertheless, it is useful to provide a catalog of common refactorings, so that developers do not need to reason directly about the definitions when evolving PLs. So in this section we illustrate different kinds of transformation templates[9] that are representative for deriving such a catalogue. We first introduce global transformations that affect a PL as a whole, changing FM, CK, and assets. Later we discuss transformations that affect PL elements in a separate and compositional way. In both cases we focus on refinement transformations, which are the essence of a refactoring catalogue, as refactoring transformations basically correspond to refinement transformations that consider quality improvement.

4.1 Overall Product Line Transformations

The first transformation we consider generalizes the refactoring illustrated by Fig. 7, where we add the optional Copy feature to an existing PL. Instead of

[9] Hereafter transformations.

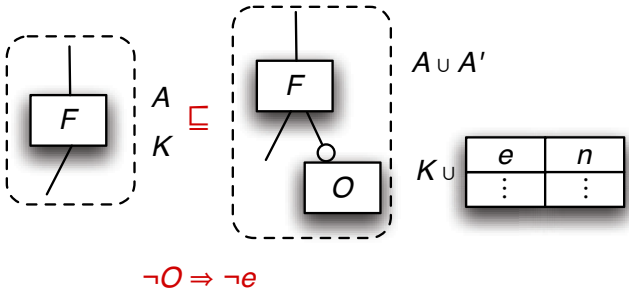

Fig. 9. Add optional feature refinement template

focusing on details of a specific situation like that, a transformation such as the one depicted by Fig. 9 precisely specifies, in an abstract way, 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. We express this by the propositional logic precondition $\neg O \Rightarrow \neg e$, which basically requires e to be equivalent to propositions of the form O or $O \wedge e'$ for any feature expression e'. This assures that products built without the new feature correspond exactly to the original PL products. 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 F. We can extend the asset mapping as wished, provided that the result is a valid asset mapping. So in this case A' should not map names that are already mapped by A. 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. For simplicity, we omit these constraints and always assume that the PLs involved in a transformation are valid.

When we need to refer to more details about the PL elements, we can use a more explicit notation, as illustrated in Fig. 10. For instance, n refers to the original set of names mapped to assets, and F now denotes the whole source FM. This establishes, in a more direct way, that it is not safe to change the original CK and asset mapping when adding an optional feature.

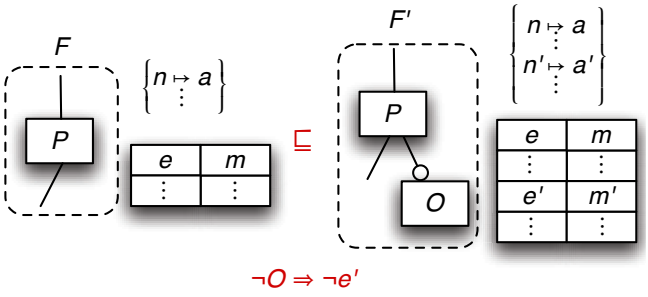

Fig. 10. Add optional feature detailed refinement template

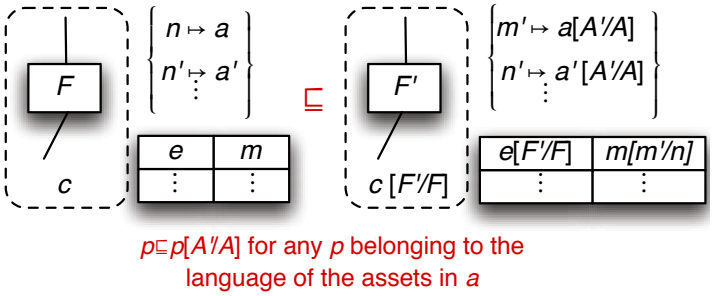

Fig. 11. PL renaming refinement template

We now discuss the PL renaming refinement transformation. Figure 11 generalizes the refactoring illustrated in Fig. 6, where we rename the Music feature and related artifacts to Audio. The transformation basically establishes that this kind of overall PL renaming is possible when renaming is a refinement in the involved core assets languages. Note that changing the name of F to F' requires changes to FM constraints ($c[F'/F]$) and CK feature expressions ($e[F'/F]$). Moreover, changing asset name n to m' requires changing the CK ($m[m'/n]$), whereas changing the asset declaration from A to A' implies changes to all declarations in a'.

Composing transformations. Besides elaborate transformations like PL renaming, which captures a major refactoring situation, it is useful to have simpler transformations that capture just one concern. In fact, from a comprehensive set of basic transformations we can, by composition, derive elaborate transformations. For example, the transformation in Fig. 12 focus on feature renaming whereas the one in Fig. 13 focus on renaming asset names, keeping the original feature names. Both do not have preconditions, so can always be applied provided the source pattern matches the source PL. By applying them in sequence and then applying asset declaration renaming (see Fig. 14), we derive the PL renaming transformation, which deals with all three concerns at once.

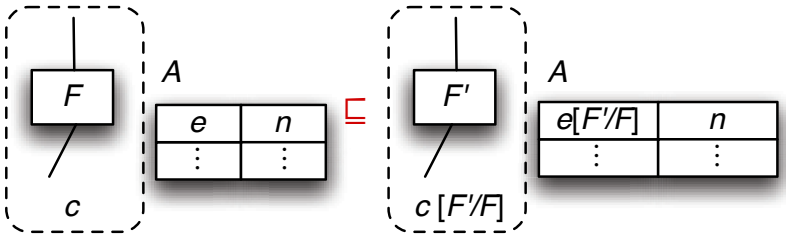

Fig. 12. Feature renaming refinement template

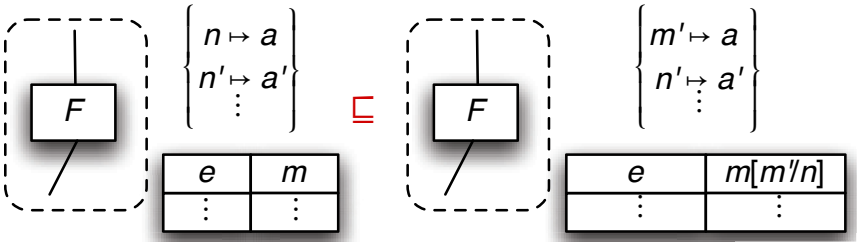

Fig. 13. Asset renaming refinement template

Whereas the elaborate transformations are useful for evolving product lines in practice, the basic transformations are useful for deriving a catalogue of elaborate transformations and verifying its soundness and completeness. It is, in fact, good practice to first propose basic transformations and then derive the elaborate ones, which are more appropriate for practical use [SB04, BSCC04].

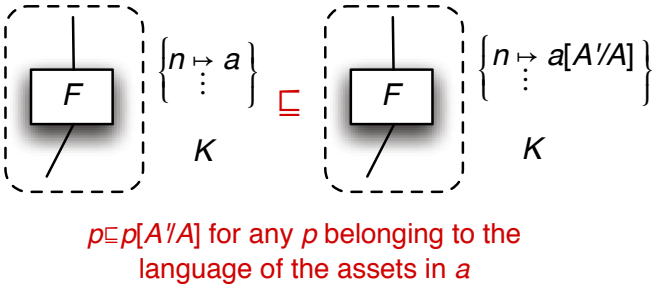

Fig. 14. Asset declaration renaming refinement template

By carefully looking at the transformation in Fig. 14, we notice that it focuses on evolving a single PL element: the asset mapping. In this case, the FM and CK are not impacted by the transformation. In general, when possible, it is useful to evolve PL elements independently, in a compositional way. So, besides the overall PL transformations illustrated so far, a refactoring catalogue should have transformations for separately dealing with FMs, CKs, and asset mappings, as illustrated in the following sections.

4.2 Asset Mapping Transformations

Transformations that focus on changing a single PL element (FM, CK, or asset mapping) should be based on specific refinements notions, not on the overall PL refinement notion. For asset mappings, exactly the same names should be mapped, not necessarily to the same assets, but to assets that refine the original ones.

Definition 10 ⟨Asset mapping refinement⟩
For asset mappings A and A', the first is refined by the second, denoted

$$A \sqsubseteq A'$$

whenever

$$\mathbf{dom}(A) = \mathbf{dom}(A') \wedge \forall a \in \mathbf{dom}(A) \cdot A(a) \sqsubseteq A'(a)$$

where $\mathbf{dom}(A)$ denotes the domain of A.

□

Note that $A(a) \sqsubseteq A'(a)$ in the definition refers to context independent asset refinement, not to program refinement.

Given this definition, we can propose transformations such as the one in Fig. 15, which renames assets declarations, provided that renaming is a refine-

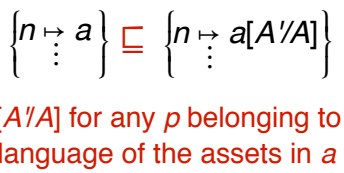

Fig. 15. Asset mapping refinement template

ment in the underlying asset languages. This essentially simplifies the transformation in Fig. 14, by focusing on changing only an asset mapping. We do not prove this here, but we can refine a PL by applying this kind of transformation and keeping FM and CK as in the source PL. This relies on the compositionality of asset refinement, as briefly discussed in Sec. 2.3, but we omit the details in order to concentrate on the different kinds of templates.

4.3 Feature Model Transformations

It is also useful to separately evolve feature models. We explore this in detail elsewhere [AGM+06, GMB08], but here we briefly illustrate the overall idea.[10] Instead of providing a refinement notion as in the previous section, we work with an equivalence. Two FMs are equivalent if they have the same semantics.

Definition 11 ⟨Feature model equivalence⟩
Feature models F and F' are equivalent, denoted $F \cong F'$, whenever $[\![F]\!] = [\![F']\!]$.

□

This equivalence is necessary for ensuring that separate modifications to a FM imply refinement for the PL as a whole. In fact, FM refinement requires only

[10] We use $\cong$ and $=$ to respectively denote semantic and syntactic equality of feature models, whereas in previous work we respectively use $=$ and $\equiv$.

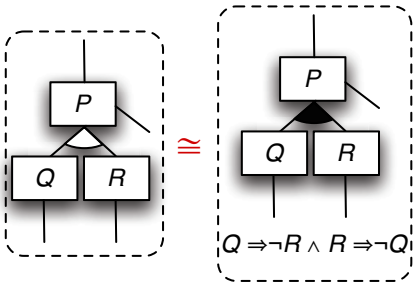

Fig. 16. Replace alternative equivalence template

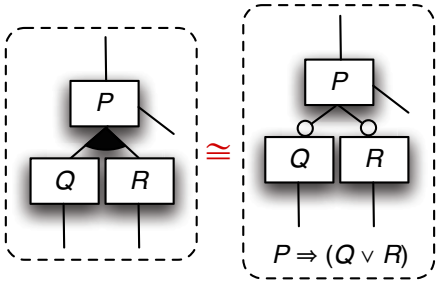

Fig. 17. Replace or equivalence template

$[\![F]\!] \subseteq [\![F']\!]$, but this allows the new FM to have extra configurations that might not generate valid programs; 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 are now 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.

Such an equivalence allows us to propose pairs of transformations (one from left to right, and another from right to left) as in Fig. 16. From left to right, we have a transformation that replaces an *alternative* by an *or* relationship, with appropriate constraints. From right to left, we have a transformation that introduces an *alternative* relationship that was indirectly expressed by FM constraints.

Similarly, Fig. 17 shows how we express *or* features as *optional* features. From left to right we can notice a pattern of transforming a FM into constraints, whereas in the opposite direction we can see constraints expressed as FM graphical notation. This is further illustrated by the transformation in Fig. 18, which, from left to right, removes *optional* relationships. A comprehensive set of transformations like these allows us to formally derive other FM equivalences, which can be useful for simplifying FMs and, consequently, refactoring a PL.

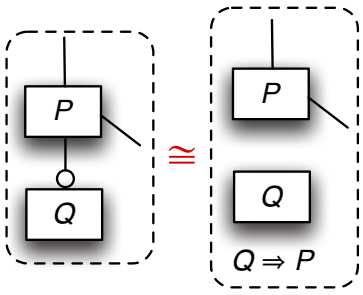

Fig. 18. Remove optional equivalence template

4.4 Configuration Knowledge Transformations

For configuration knowledge transformations, we also rely on an equivalence relation.

Definition 12 ⟨Configuration knowledge equivalence⟩
Configuration knowledge K is equivalent to K', denoted $K \cong K'$, whenever $[\![K]\!] = [\![K']\!]$. □

Due to the equivalence, we again have pairs of transformations. Figure 19 illustrates four such transformations, indicating that we can change rows in a CK (bottom left), apply propositional reasoning to feature expressions (top left), merge rows that refer to the same set of asset names (top right), and merge rows with equivalent expressions (bottom right). This is not a complete set of transformations, but gives an idea of what a comprehensive PL refactoring catalogue should contain.

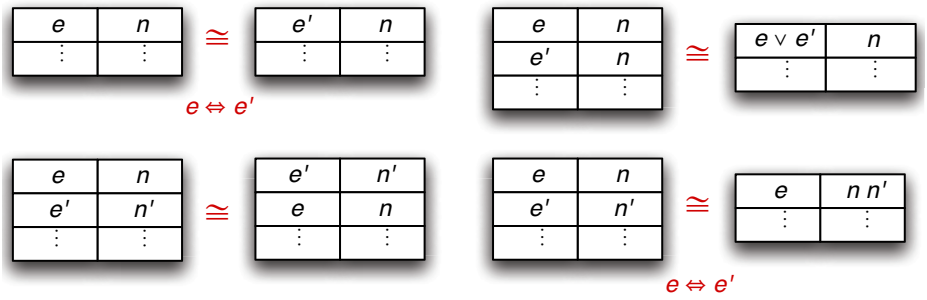

Fig. 19. Configuration knowledge equivalence templates

5 Conclusions

In this chapter we introduce and formalize notions of refinement and refactoring for software product lines. Based on these notions and specific ones for feature

models, configuration knowledge, and asset mappings, we also illustrate the kinds of transformation that should constitute a product line refactoring catalogue.

We hope this stimulates further work towards such a catalogue, considering different notations and semantic formalizations for software product line artifacts [CHE05, Bat05, SHTB07, GMB08]. This is important to guide and improve safety of the product line derivation and evolution processes. The presented transformation templates precisely specify the transformation mechanics and preconditions. This is especially useful for correctly implementing these transformations and avoiding typical problems with current program refactoring tools [ST09]. In fact, even subtler problems can appear with product line refactoring tools.

The ideas formalized here capture previous practical experience on product line refactoring [ACV+05, AJC+05, TBD06, KAB07], including development and use of a product line refactoring tool [CBS+07, ACN+08]. However, much still has to be done to better evaluate our approach and adapt existing processes and tools for product line refactoring.

Besides working towards a comprehensive refactoring catalogue, we hope to formally prove soundness and study completeness of product line transformations for the notations we use here for feature models and configuration knowledge. We should also discuss refinement and refactoring properties, which we omitted from this introduction to the subject.

Acknowledgements

I would like to thank the anonymous reviewers and colleagues of the Software Productivity Group for helping to significantly improve this work. Leopoldo Teixeira and Márcio Ribeiro provided interesting bad smells and refactoring scenarios from the Mobile Media example. Rodrigo Bonifácio played a fundamental role developing the configuration knowledge approach used here. Vander Alves, Rohit Gheyi, and Tiago Massoni carried on the initial ideas about feature model and product line refactoring. Fernando Castor, Carlos Pontual, Sérgio Soares, Rodrigo, Leopoldo, Rohit, and Márcio provided excellent feedback on early versions of the material presented here. Besides all, working with those guys is a lot of fun! I would also like to acknowledge current financial support from CNPq, FACEPE, and CAPES projects, and early support from FINEP and Meantime mobile creations, which developed the Rain of Fire product line.

References

[ACN+08] Alves, V., Calheiros, F., Nepomuceno, V., Menezes, A., Soares, S., Borba, P.: FLiP: Managing software product line extraction and reaction with aspects. In: 12th International Software Product Line Conference, p. 354. IEEE Computer Society, Los Alamitos (2008)

[ACV+05] Alves, V., Cardim, I., Vital, H., Sampaio, P., Damasceno, A., Borba, P., Ramalho, G.: Comparative analysis of porting strategies in J2ME games. In: 21st IEEE International Conference on Software Maintenance, pp. 123–132. IEEE Computer Society, Los Alamitos (2005)

[AGM+06] Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., Lucena, C.: Refactoring product lines. In: 5th International Conference on Generative Programming and Component Engineering, pp. 201–210. ACM, New York (2006)

[AJC+05] Alves, V., Matos Jr., P., Cole, L., Borba, P., Ramalho, G.: Extracting and evolving mobile games product lines. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 70–81. Springer, Heidelberg (2005)

[Bat05] Batory, D.: Feature models, grammars, and propositional formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)

[BB09] Bonifácio, R., Borba, P.: Modeling scenario variability as crosscutting mechanisms. In: 8th International Conference on Aspect-Oriented Software Development, pp. 125–136. ACM, New York (2009)

[BSCC04] Borba, P., Sampaio, A., Cavalcanti, A., Cornélio, M.: Algebraic reasoning for object-oriented programming. Science of Computer Programming 52, 53–100 (2004)

[CB05] Cole, L., Borba, P.: Deriving refactorings for AspectJ. In: 4th International Conference on Aspect-Oriented Software Development, pp. 123–134. ACM, New York (2005)

[CBS+07] Calheiros, F., Borba, P., Soares, S., Nepomuceno, V., Alves, V.: Product line variability refactoring tool. In: 1st Workshop on Refactoring Tools, pp. 33–34 (July 2007)

[CE00] Czarnecki, K., Eisenecker, U.: Generative programming: methods, tools, and applications. Addison-Wesley, Reading (2000)

[CHE05] Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their specialization. Software Process: Improvement and Practice 10(1), 7–29 (2005)

[CN01] Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Reading (2001)

[FCS+08] Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Filho, F., Dantas, F.: Evolving software product lines with aspects: an empirical study on design stability. In: 30th International Conference on Software Engineering, pp. 261–270. ACM, New York (2008)

[Fow99] Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading (1999)

[GA01] Gacek, C., Anastasopoulos, M.: Implementing product line variabilities. SIGSOFT Software Engineering Notes 26(3), 109–117 (2001)

[GMB05] Gheyi, R., Massoni, T.M., Borba, P.: A rigorous approach for proving model refactorings. In: 20th IEEE/ACM International Conference on Automated Software Engineering, pp. 372–375 (2005)

[GMB08] Gheyi, R., Massoni, T., Borba, P.: Algebraic laws for feature models. Journal of Universal Computer Science 14(21), 3573–3591 (2008)

[KAB07] Kastner, C., Apel, S., Batory, D.: A case study implementing features using AspectJ. In: 11th International Software Product Line Conference, pp. 223–232. IEEE Computer Society, Los Alamitos (2007)

[KCH+90] Kang, K., Cohen, S., Hess, J., Novak, W., Spencer Peterson, A.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (1990)

[KHH+01] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: Getting started with AspectJ. Communications of the ACM 44(10), 59–65 (2001)

[Kru02] Krueger, C.: Easing the transition to software mass customization. In: van der Linden, F.J. (ed.) PFE 2002. LNCS, vol. 2290, pp. 282–293. Springer, Heidelberg (2002)

[MGB08] Massoni, T., Gheyi, R., Borba, P.: Formal model-driven program refactoring. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 362–376. Springer, Heidelberg (2008)

[Opd92] Opdyke, W.: Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign (1992)

[PBvdL05] Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Heidelberg (2005)

[Rob99] Roberts, D.: Practical Analysis for Refactoring. PhD thesis, University of Illinois at Urbana-Champaign (1999)

[SB04] Sampaio, A., Borba, P.: Transformation laws for sequential object-oriented programming. In: Cavalcanti, A., Sampaio, A., Woodcock, J. (eds.) PSSE 2004. LNCS, vol. 3167, pp. 18–63. Springer, Heidelberg (2006)

[SHTB07] Schobbens, P.-Y., Heymans, P., Trigaux, J.-C., Bontemps, Y.: Generic semantics of feature diagrams. Computer Networks 51(2), 456–479 (2007)

[ST09] Steimann, F., Thies, A.: From public to private to absent: refactoring java programs under constrained accessibility. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 419–443. Springer, Heidelberg (2009)

[TBD06] Trujillo, S., Batory, D., Diaz, O.: Feature refactoring a multi-representation program into a product line. In: 5th International Conference on Generative Programming and Component Engineering, pp. 191–200. ACM, New York (2006)

[vdLSR07] van der Linden, F., Schmid, K., Rommes, E.: Software Product Lines in Action: the Best Industrial Practice in Product Line Engineering. Springer, Heidelberg (2007)

[vO02] van Ommering, R.C.: Building product populations with sofwtare components. In: 24th International Conference on Software Engineering, pp. 255–265. ACM, New York (2002)