Investigating the Safe Evolution of Software Product Lines

Laís Neves, Leopoldo Teixeira, Paulo Borba
Informatics Center
Federal University of Pernambuco
50740-540, Recife – PE – Brazil
{lmn3, lmt, phmb}@cin.ufpe.br

Vander Alves
Computer Science Department
University of Brasília
70910-900, Brasília – DF – Brazil
valves@unb.br

Demóstenes Sena, Uirá Kulesza
Computing Department
Federal University of Rio Grande do Norte
59072-970, Natal – RN – Brazil
demostenes.sena@ifrn.edu.br, uira@dimap.ufrn.br

Abstract

The adoption of a product line strategy can bring significant productivity and time to market improvements. However, evolving a product line is risky because it might impact many products and their users. So when evolving a product line to introduce new features or to improve its design, it is important to make sure that the behavior of existing products is not affected. In fact, to preserve the behavior of existing products one usually has to analyze different artifacts, like feature models, configuration knowledge and the product line core assets. To better understand this process, in this paper we discover and analyze concrete product line evolution scenarios and, based on the results of this study, we describe a number of safe evolution templates that developers can use when working with product lines. For each template, we show examples of their use in existing product lines. We evaluate the templates by also analyzing the evolution history of two different product lines and demonstrating that they can express the corresponding modifications and then help to avoid the mistakes that we identified during our analysis.

Categories and Subject Descriptors D.2.7 [*Distribution, Maintenance, and Enhancement*]: Restructuring, reverse engineering, and reengineering; D.2.13 [*Reusable Software*]: Domain engineering

General Terms Design, Languages

Keywords Software Product Lines, Refinement, Product Line Safe Evolution

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. This kind of reuse targeted at a specific set of products can bring significant productivity and time to market improvements [21, 25]. To obtain these benefits with reduced upfront investment and risks, previous work [2, 7, 18] proposes to minimize the initial product line (domain) analysis and development process by deriving a product line from an existing product. A similar process applies to evolving a product line, when adding new products or improving the product line design requires extracting variations from previous parts shared by a set of products.

Manually extracting and changing different code parts when evolving a product line requires substantial effort, especially for checking necessary conditions to make sure the extraction is correctly performed. Moreover, this process is tedious and can also introduce defects, modifying the behavior of the products before the extraction process, and compromising the promised benefits on other dimensions of costs and risks.

To better understand this process, in this paper we discover and analyze concrete product line evolution scenarios and, based on the results of this study, we describe a number of safe evolution templates that developers can use when working with product lines. For this, we rely on a product line refinement notion that preserves the product line original products behavior [5]. This notion is important for safely evolving a product line by simply improving its design or even by adding new products while preserving existing ones, assuring safety for product line existing users. We use the definitions and suggestions from our group previous works [4], [5] to identify evolution scenarios and then generalize these scenarios to other situations through the templates.

These templates specify transformations that go beyond program refactoring notions [14, 22] by considering both sets of reusable assets that not necessarily correspond to valid programs, and extra artifacts, such as feature models (FM) [10, 15] and configuration knowledge (CK) [10], which are necessary for automatically generating products from assets. Previous work [1] from our group describes a set of product line refactorings regarding feature models. We extend this work describing now safe evolution templates that also deal with configuration knowledge and product line assets, in addition to feature model.

For each template, we show examples of their use in existing product lines. We evaluate the templates by also analyzing the evolution of two different product lines and demonstrating that they can express the corresponding modifications and then help to avoid the mistakes that we identified during our analysis.

This paper makes the following contributions:

- we discover and analyze product line evolution scenarios by mining part of a product line SVN history (Section 2);
- we identify and describe precisely a number of product line safe evolution templates that abstract, generalize and factorize the analyzed scenarios (Section 4);
- we show evidence that the identified templates can justify all evolution scenarios identified in the SVN history of two product lines and could avoid the mistakes that we found during our

GPCE'11, October 22–23, 2011, Portland, Oregon, USA.

analysis. We also show the frequency of use for each template in the analyzed scenarios (Section 5);

Besides these sections, Section 6 lists the works related to our research and Section 7 presents the concluding remarks.

2. Motivating Example

In order to better illustrate the problems that might occur when manually evolving PLs, we present a maintenance scenario based on TaRGeT PL [12]. TaRGeT is a PL of automatic test generation tools and is implemented using Eclipse RCP plug-ins technology. TaRGeT has been developed since 2007 by our research group and its current version has 42 implemented features and counts approximately 32,000 LOC. The history track of 3 major releases and 10 minor releases is available in an SVN repository.

While analyzing TaRGeT's SVN history we have found several evolution steps that were supposed to be safe, such as a design improvement or the addition of new products, but actually introduced errors to the PL. **Scenario A** describes one of these cases, the implementation of the new *Component Keyword* text field in the *Test Case Preferences* window. Figure 1 shows the new field. This field should only appear when the *XLS STD* feature is selected. This feature is related to TaRGeT's output format and, when selected, generates test suites in a format compatible with Microsoft Excel.

With that in mind, Figure 2 describes changes applied to the FM, CK and source code artifacts in order to address this evolution scenario. The CK notion that we use here is represented as a table and maps feature expressions (in the left-hand side column) to asset names (in the right-hand side column). We use propositional formulae having feature names as atoms to represent feature expressions that represent presence conditions [11]. In this case, due to limited space, we only show fragments of CK and FM that are related to the example context. As TaRGeT has many features implemented, due to limited space it is not possible to show everything here. The tick signs indicate what changed.

In summary, to modify the PL, the developers first created the aspect *STDPreferencesAspect*, which is responsible for introducing the *Component Keyword* field. However, when updating the CK, they made a mistake and the aspect was associated to the *Output* feature instead of *XLS STD*. The developers then tested product with the *XLS STD* feature selected and saw that the new field was present in the window as expected. They also tested the products with other output formats but they only verified if the main features were correct. As a consequence, they thought that the products were working as expected and committed the modified code to make the changes effective. They did not notice that the new *Component Keyword* field became visible in all products, thus introducing a bug in the PL.

This example demonstrates that manually evolving a PL is error-prone, because in order to make sure that the behavior of existing products is not affected, one usually has to analyze different artifacts, like FMs, CK and the PL assets (such as classes, configuration files or aspects). In addition, the bugs that might be introduced during manual evolution of PL could be difficult to track because they are present only in certain product configurations and the configuration space normally increases.

Analyzing TaRGeT's evolution history through releases 4.0 to 6.0 (from January 2009 to January 2010), we identified a total of 20 evolution scenarios. We provide more details about how we identified these scenarios in Section 5. We verified that the minimum number of modified classes in these scenarios was 1 and the maximum was 54, and the average number of modified classes was 12.9. We also found that about 10% of these modifications introduced a defect in the PL. This shows that issues like the one just presented might happen often and demand special attention.

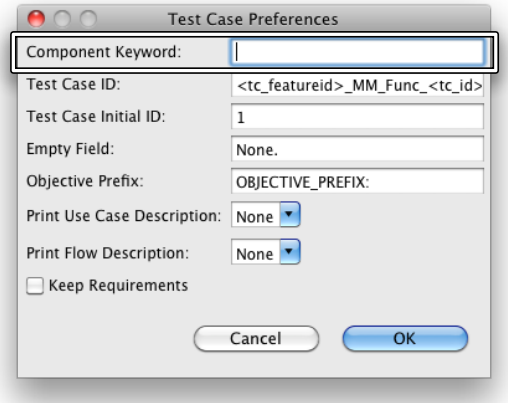

Figure 1. Scenario A - *Test Case Preferences* widow with the new *Component Keyword* field

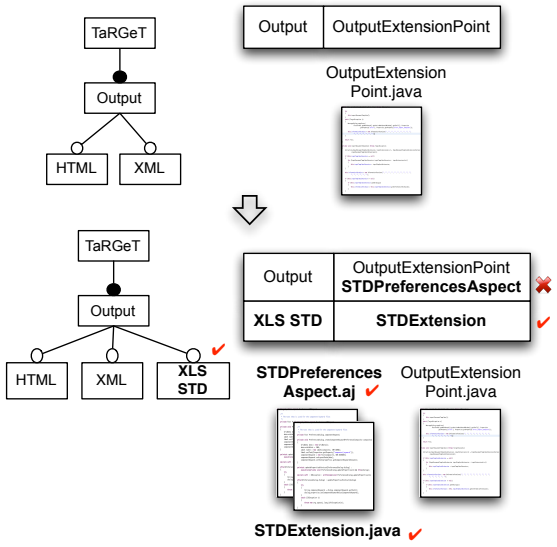

Figure 2. Scenario A - Adding the new optional feature *STD Output*

To better understand the problems that might occur with manual PL evolution, in following sections we explain how we discovered and analyzed concrete PL evolution scenarios and, based on the results of this study, we describe a number of safe evolution templates that developers can use when working with PLs.

3. Product Line Refinement and Safe Evolution

To guide our PL evolution analysis and help us to identify the evolution scenarios, we rely on a PL refinement notion [4, 5]. Such notation is based on a program refinement notion [5], which is useful for comparing assets with respect to behavior preservation. In this section we briefly introduce these necessary concepts to understand the PL safe evolution templates described in the next section.

The formal definition for PLs consists of a FM, a CK, and an asset mapping (AM) that jointly generate products, that is, well-formed asset sets in their target languages. The set of all valid product configurations corresponds to the semantics of a feature model and we represent it as $[\![F]\!]$. We represent the application of the semantics function to a configuration knowledge K, an asset mapping A, and a configuration c as $[\![K]\!]^A_c$. This function maps product configurations and AMs into finite sets of assets (products).

DEFINITION 1. ⟨Product line⟩
For a feature model F, an asset mapping A, and a configuration knowledge K, we say that tuple

$$(F, A, K)$$

is a product line when, for all $c \in [\![F]\!]$,

$$wf([\![K]\!]^A_c)$$

□

The wf represents the well-formedness constraint and 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. Here we demand PL elements to be coherent as explained.

Similar to program refinement, PL refinement preserves behavior. However, it goes beyond source code and other kinds of reusable assets, and might transform FM and CK as well. The notion of behavior is 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. This is exactly what guarantees safety when improving the design of a PL.

In most of PL refinement scenarios, however, many changes need to be applied to the code assets, FMs and configuration knowledge, which often [18] leads the refactored PL to generate more products than before. 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. This is illustrated by Figure 3, where we refine the simplified MobileMedia PL (detailed in Section 5) by adding the optional Copy feature. 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.

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

DEFINITION 2. ⟨Product line refinement⟩
For product lines (F, A, K) and (F', A', K'), the second refines the first, denoted

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

whenever

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

□

Remember that, for a configuration c, a configuration knowledge K, and an asset mapping A related to a given product line, $[\![K]\!]^A_c$ is a well-formed set of assets. Thus $[\![K]\!]^A_c \sqsubseteq [\![K']\!]^{A'}_{c'}$ refers to asset set refinement. The definition just mentioned relates two PLs, therefore, all products in the target PL should be well-formed. The PL refinement definition is a pre-order, which allows us to compose different safe evolution templates and also have a valid well-formed PL. The definition is also compositional, in the sense that refining one of FM, AM, or CK that are part of a valid PL yields a refined valid PL. Such a compositionality property is essential to guarantee independent development of these artifacts in a PL.

The $\sqsubseteq$ symbol is a relation that says when the transformation is safe. It does not mean functional equivalence because the PL refinement notion is a pre-order. It can reduce non-determinism, for example.

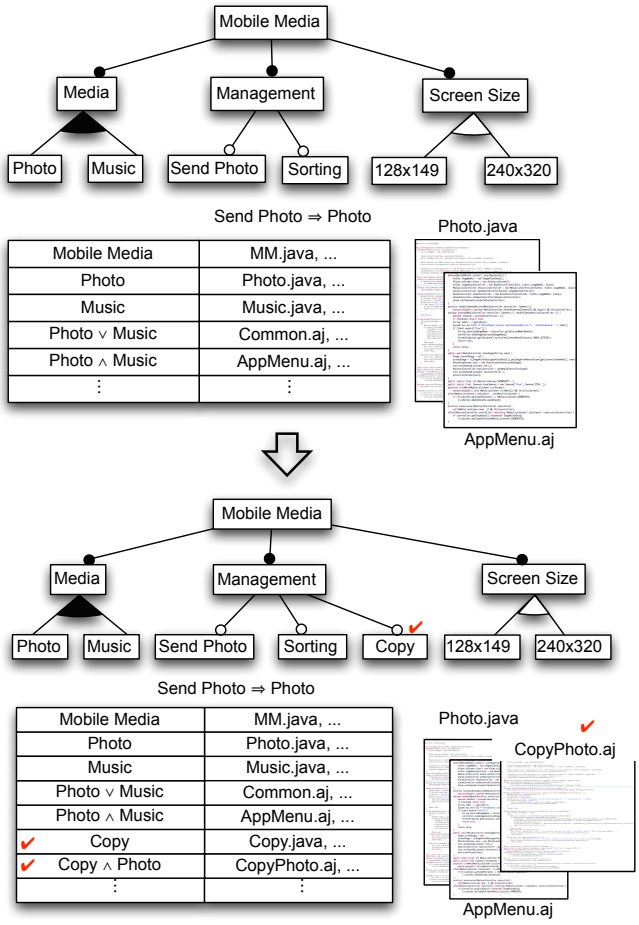

Figure 3. Adding an optional feature refinement

4. Safe Evolution Templates for PLs

Based on this notion of refinement, in this section, we describe how we discovered and analyzed concrete PL evolution scenarios and, from the results of this study, we present a number of safe evolution templates. The templates provide guidance on how to structure extracted variant parts and help to avoid problems that might occur when evolving PLs manually.

The templates are valid modifications that can be applied to a PL thereby improving its quality and preserving existing products' behavior. The PL transformations listed here involve artifacts like FMs, CK and core assets, which can be source files, aspects, configuration files and others. It is important to mention that the PL refinement notion that we rely on (see Section 3) is independent from the FM and CK [5] languages used. However, as the safe evo-

lution templates describe operations using these artifacts, they are specific to the language used to define them.

To discover the safe evolution templates, we identified and analyzed different evolution scenarios from the TaRGeT PL between releases 4.0 to 5.0. During this time, we identified a total of 11 safe evolution scenarios, which means evolution steps according to the refinement notion that we rely on. After this step, we analyzed the changes performed in code assets, FM and CK. We also considered SVN commit comments and revision history annotations in the source files. Based on these results, we derived a set of safe evolution templates that abstract, generalize and factorize the analyzed scenarios and can be used in different contexts.

Regarding the used notation, each template described here shows the FM, AM and CK status before and after the transformation. FMs contain only the features that are necessary to understand the templates. These are the features that are involved or are affected by the template being described. If we want to say that a feature may contain other features related to it, this can be expressed by a trace above or below the feature.

We represent the CK with a two-column table, in which the left-hand column contains feature expressions, which represent presence conditions **??**, that are mapped to asset names, represented in the right-hand column. This representation is useful to separate asset information from its configuration. The ellipsis indicates other lines different from the one that is explicitly expressed. An asset mapping maps asset names into assets. It is useful to avoid conflicting assets names in the CK. Two curly braces represent the AM, grouping a mapping of asset names, on the left-hand side, to assets, on the right-hand side.

In the case that FM, AM or CK are not changed in the transformation, they are expressed in the template by the single letters *F*, *A* and *K*, respectively. Each template also declares meta-variables that abstractly represent the PL elements, for example an arbitrary feature expression or an arbitrary asset name. The letters *F*, *A* and *K* are also meta-variables. If a variable appears in both sides of a transformation, this means that it remains unchanged. On the bottom we express the pre-conditions to the transformation.

4.1 Template 1 – Split Asset

When analyzing evolution scenarios that changed a mandatory feature to optional, we observe that this type of operation usually involved tracking the code related to the feature and extracting it to other artifacts, like aspects, property files or Eclipse RCP plug-in extensions (in case of TaRGeT).

To generalize these cases, we derived Template 1, illustrated by Figure 4. This template indicates that it is possible to split an asset *a* into two other assets *a'* and *a"* as long as the set of assets *a'a"* refines asset *a*. The first restriction is necessary to guarantee that the behavior of the old asset *a* is preserved. We used the other restrictions to simplify CK and AM representation. When *n* appears in other CK lines, it is only necessary to change all occurrences for *n*, *n'*.

Another variation of this template is that *n'* can also represent an existing asset in the PL. We could discuss other variations too, but the focus here is on the basic template. The variations can often be derived from the basic template by composing it with other templates. We can say that Template 1 is a PL refinement because for each product that contained asset *a* in the source PL, there is now a corresponding refined one in the target PL that contains the composition of assets *a'* and *a"*. This specific transformation improves PL quality because it is possible to modularize feature behavior in different assets.

In our study, this template was used, in the scenario that appears in Figure 5, where developers want to extract the *Interruption* feature code that is scattered along three different Java classes (the

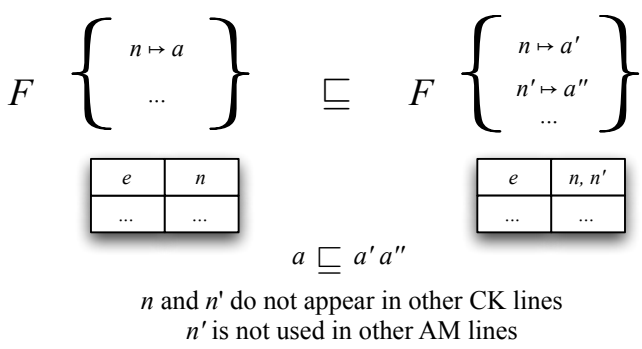

Figure 4. Template 1 - Split asset

ones listed in the CK) to a new Aspect. For this, they applied the *Split Asset* template and its variations several times and extracted the feature code to the new aspect *PMInterruptionAspect*. The FM is not affected by the transformation. The source files in the figure represents the AM.

We decided to present the templates in details because we intend to provide tool support for them in the future. So, this details will be important when defining the tool requirements. We use a declarative approach to describe the templates because we believe this improves presentation and helps developers to better understand the templates. Traditional approaches like the used in Fowler's templates are imperative and difficult presentation.

The PL refinement notion that supports our templates is not limited to any specific kind of asset, which can be class files, aspects, configuration files, among others. That is why we can use them in different contexts from the ones we present in this work. However, some of our templates have restrictions associated to assets refinement, as for example in Template 1. In order to address these restrictions, the developer should choose a proper asset refinement notion that can be more or less restrictive, such as programming transformation laws or tests, according to his reality.

4.1.1 Split Asset – Code Transformations

In order to better explain the code transformations, such as in the example presented on Figure 5, we need transformations that deal specifically with code assets. This is necessary because the general abstract PL templates only establishes the refinement constraint. That is why we specify more precisely code transformations templates that complement the general templates for PL safe evolution. The existing refactorings in the Eclipse IDE are an example of these code transformations to refactor Java code assets.

For Template 1, there are many variation extraction mechanisms described in other works [2, 8] that could be classified as code transformation templates. However, we only mention here the ones that we observed in our analysis of TaRGeT PL. These code templates are helpful because they propose valid transformations that do not deteriorate the PL. So if a developer needs to maintain the PL, with the templates he reduces the possibility of introducing errors, increasing confidence.

In a practical scenario, like the one that appears in Figure 5, the developer first selects a PL template that makes the necessary transformation in the CK and FM, in this case the Split Asset template, and then selects the code transformation templates to actually implement the necessary changes in code assets. In the above mentioned example, we applied templates *Extract Resource to Aspect - after*, *Extract Method to Aspect*, *Extract Context* and *Extract After Block* to extract the feature code to the *PMInterruptionAspect*. These code templates first appeared in a catalog of refactoring tem-

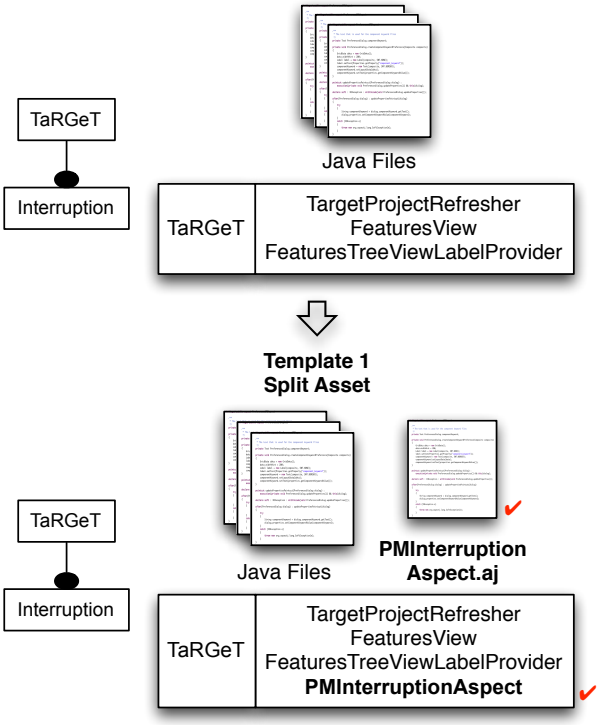

Figure 5. Split asset example

plates to extract code from classes to aspects, using AspectJ [2]. The templates from this catalog rely on aspect oriented programming to modularize crosscutting concerns, which often occur in PLs.

Another code template that we identified during our analysis is useful to extract constants in the source code, usually user interface texts, to a properties file. We could define many other code templates to other types of values. This operation is commonly performed when there is the need to localize the user interface to support different languages, like in the example shown in Figure 6, where the form title text *"Test Selection Page"* is moved to a properties file. In this code template the original class is refined by the composition of the new refactored class with a call to the properties file, and the properties file itself.

Figure 7 shows the abstract transformation template. The notation used follows the representation of programming laws [6]. On the right-hand side, all occurrences of text *s* in *body* are replaced by a call to the property that contains its value. We denote the set of field declarations, method declarations and properties declarations by *fs*, *ms* and *ps*, respectively. We use *T* to represent the return type of method *m*. In class *C* constructor we place a call to a new method responsible to load the properties file. On the bottom we list the transformation restrictions.

In our study, we also identified code transformation scenarios that involved variation extraction to extension points. To capture that, we have a code template that represents this operation. This template uses Eclipse plug-ins extension point pattern and defines that it is possible to extract code within a class, create an extension point and replace the code in the class by a call to existing extensions that implement that new extension point. A new extension is then created with the extracted code. Due to limited space, we do not show the transformation details. As TaRGeT is an Eclipse RCP application, we observed that many variations are implemented

```
public void createFormContent(IManagedForm managedForm) {

    ScrolledForm form = managedForm.getForm();
    form.setText("Test Selection Page");

}
```

```
public void createFormContent(IManagedForm managedForm) {

    ScrolledForm form = managedForm.getForm();
    form.setText(properties.getProperty("test_selection_page"));

}
```

+

```
invalid_similarity_value = Invalid Similarity Value
test_purpose_creation = Test Purpose Creation
test_selection_page = Test Selection Page
```

Figure 6. Example of text extraction to properties file

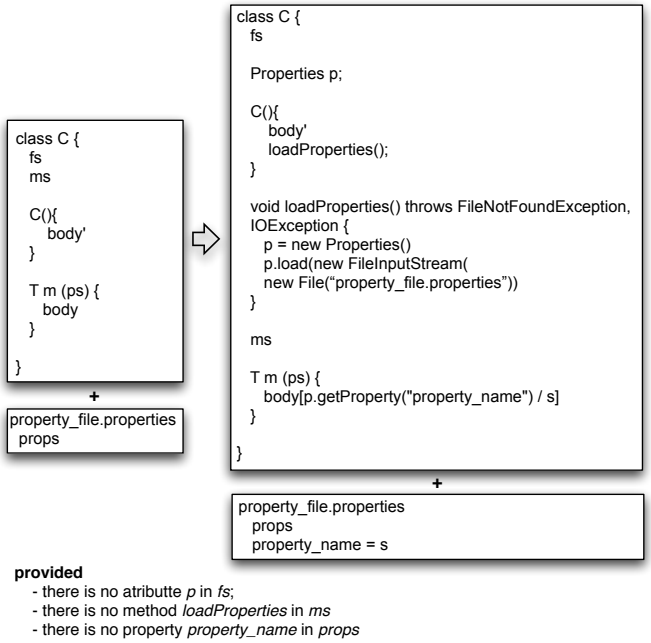

Figure 7. Abstract template for text extraction to properties file

with the extension point mechanism and can be justified by this template.

4.2 Template 2 – Refine Asset

Another template that we propose based on the observations of our study is shown in Figure 8. This template defines that it is possible to modify an asset *a*, transforming it into asset *a'*, as long as the new asset *a'* refines the original asset *a*. The first restriction is related to asset set refinement. We assure PL refinement because the behavior of each product that contained asset *a* is preserved by a corresponding refined product that contains asset *a'*.

Template 2 also relies on code transformation templates. For example, we could use Template 2 combined with existing refactorings for object-oriented, aspect-oriented, and conditional compilation programs, for example Eclipse refactorings. In practice, we know that some of these code transformations might change other

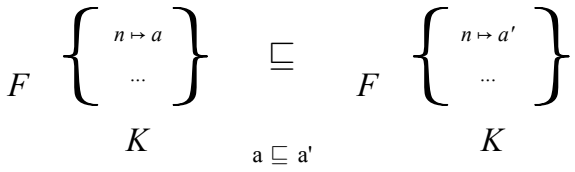

Figure 8. Template 2 - Refine asset

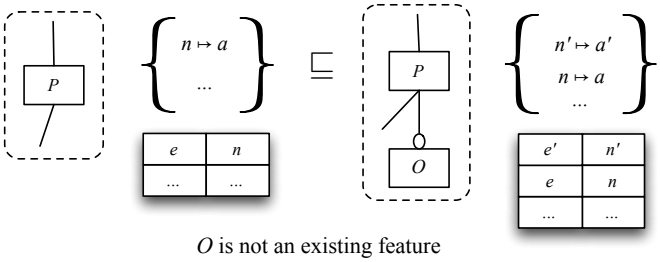

Figure 9. Template 3 - Add new optional feature

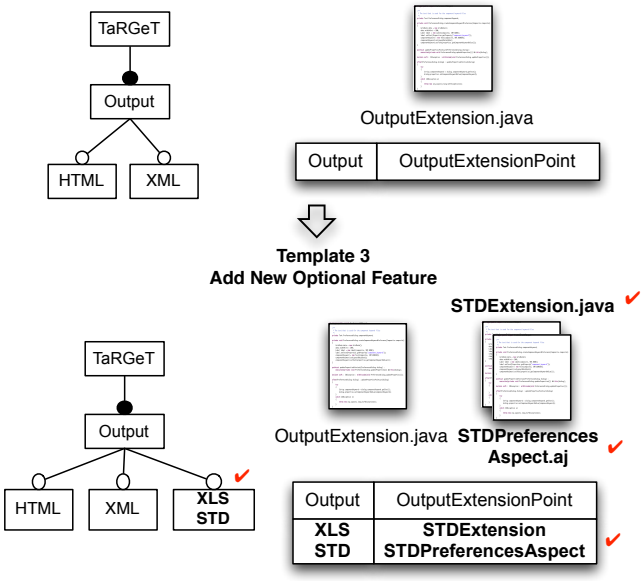

Figure 10. Add new optional feature example

code assets. For instance, if a class that is used by other classes is renamed (class-renaming refactoring), these classes need to be modified as well. We can have variations of Template 2 to deal with these cases that involve more than one asset.

4.3 Template 3 – Add New Optional Feature

Template 3 emerged when analyzing evolution scenarios like the one described in our motivating example, when a developer needs to introduce an optional feature to the PL. This template, presented in Figure 9, states that it is possible to introduce a new optional feature *O* and add a new asset *a* associated to a feature expression *e'* in the CK only if the restriction that says that selecting *e'* implies selecting *O* is respected. The restriction assures that the new assets are only present in products that have feature *O* selected and that products built without the new feature correspond exactly to the original PL products. The template also states that we can not have another feature named *O* in the FM nor another asset name *n'* in the AM, and that the resulting PL is well-formed. We assure refinement because the resulting PL has the same products that it had before in addition to products that contain feature *O*, and we improve the resulting PL quality by increasing its configurability.

In practice, the template implementation should be flexible enough to allow the association of more than one asset to the new optional feature in the CK. Figure 10 shows the application of Template 3 in the scenario that we describe in our motivating example. The new assets *STDExtension* and *STDPreferencesAspect* are correctly associated to the new *XLS STD* feature in the CK.

4.4 Template 4 – Add New Mandatory Feature

Template 4, represented in Figure 11, indicates that we can insert a new mandatory feature, represented by *M*, on a FM as long as we preserve the CK, represented by *K*, and the AM, represented by *A*. The template restricts that we can not have another feature named *M* in the FM. This transformation is a refinement because the AM and the CK do not change, so products before and after the transformation are still the same. It increases quality because it improves FM readability.

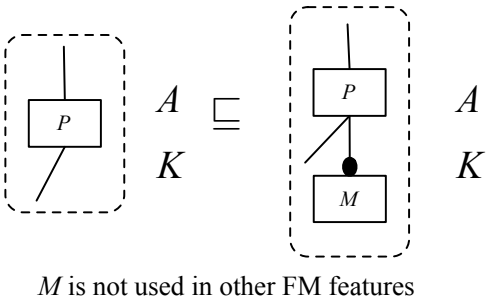

Figure 11. Template 4 - Add new mandatory feature

If we had added new assets and associated them with feature *M* in the CK, it would not be possible only with this template to ensure that these artifacts would not alter the behavior of existing products. Consequently, we could not assure that the transformation was a safe evolution step. It is possible to generalize even more the template considering that we can add any kind of feature to the FM (mandatory, optional, alternative, or) if we preserve code assets and CK.

Figure 12 illustrates the template utilization. In this example, developers inserted a new mandatory feature *Word* under the *Input* feature. This feature identifies the possible formats of use case documents that TaRGeT accepts as input. The *Word* feature represents the Microsoft Word format. This transformation is useful to improve FM readability and also to restructure the PL to receive new features, as demonstrated in Figures 12 and 14. Similarly to this operation, it is possible to add any kind of feature (optional, alternative, or) in the FM, as long as the CK and FM do not change.

We observed in our study that Template 4 is usually used together with other templates following it. We decided to divide this transformation into two steps to facilitate the automation and reuse of the templates, since it is possible to combine templates to derive

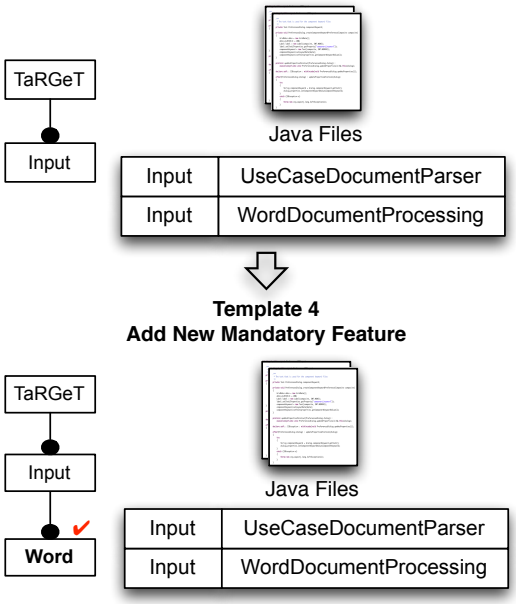

Figure 12. Add new mandatory feature example

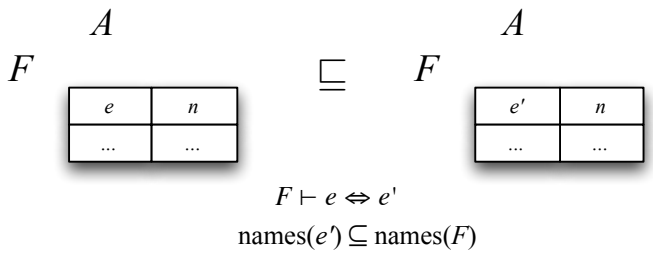

Figure 13. Template 5 - Replace feature expression

more complex transformations, due to the transitivity of the PL refinement notion.

4.5 Template 5 – Replace Feature Expression

Template 5 in Figure 13 expresses that it is possible to change the feature expression associated to an asset *n* in the CK from *e* to *e'*, when these expressions are equivalent according to the FM. The restrictions specify that all product configurations from a feature model *F* lead to equivalent evaluation for the feature expressions in both *e* and *e'*. They also specify that the feature expression *e'* only references names from *F*. This template improves PL quality by enhancing CK readability.

We found many occurrences of this template combined with Template 4. Figure 14 illustrates how we can use Template 5 in the example described in Figure 12. In this example, the template changed the feature expression related to the *WordDocumentProcessing* asset from *Input* to *Word*. This is possible because as *Word* is a child of *Input*. Thus, selecting the first means that the second is also selected. This operation is useful because it improves CK readability, as it allows to associate asset names to more coherent feature expressions. It also helps to restructure PL existing features, like *Word* and *Input* in Figure 12 example.

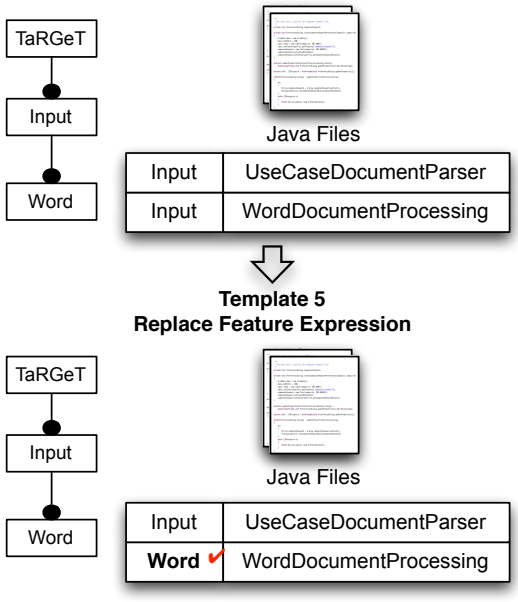

Figure 14. Template 5 - Replace feature expression example

4.6 Discussion

When deriving the templates we assured first that they complied with the refinement notion that we rely on. After deriving the preliminary versions, we realized that in some cases the restrictions associated to the templates were too strong and that they could not be used in other situations different from the ones we analyzed. So we decided to discard unnecessary conditions in order to make the templates more general and consequently more useful.

Besides, we found that we could divide some templates into more steps, which improved understanding and would help an automation process in the future in order to provide better support to developers. So we refactored and evolved these templates and new ones were derived.

We also observed that in some evolution scenarios that we analyzed, it was usually necessary to combine more than one template, for example, templates 4 and 5. This information can be useful when defining a strategy to compose the templates in an automated solution. The transitivity of the PL refinement notion allows the combination of different templates resulting in a well-formed PL.

A limitation of our templates it that they are not able to support preprocessor based annotations, which we can implement in Java using third party tools like Velocity. This is because the CK notation that we use here maps feature expressions to asset names, which are described in the AM. In order to support these annotations, it would be necessary to extend the CK notion, allowing mapping asset transformations into assets. We intend to this as future work.

Table 1 summarizes all templates proposed in this work. Templates 1 to 5 are detailed in this section. Template 3 was first mentioned in a previous work [4]. Template 6 defines transformations that occur in CK and AM when adding a new alternative feature in the FM. Similarly, Template 7 defines the same transformations to include an OR feature in the FM. Finally, Template 8 defines asset removal and contains the program transformation law for class elimination [6] defined as a code transformation template. We derived this template when analyzing another PL, which we describe

Table 1. Safe Evolution Templates for Product Lines

Template	Name
1	Split Asset
2	Refine Asset
3	Add New Optional Feature
4	Add New Mandatory Feature
5	Replace Feature Expression
6	Add New Alternative Feature
7	Add New OR Feature
8	Delete Asset

in more details in Section 5. The templates not detailed here can be found in our website.[1]

5. Analyzing Product Line Safe Evolution

This section describes how we investigated the evolution scenarios and presents an analysis on the expressivity of our safe evolution templates. We also show the frequency of use of each template in the evolution scenarios. Finally, we list the main threats to our study in the end of this section.

5.1 Data and Setup

To perform our study, we chose two different PLs. We first studied TaRGeT [12], which we previously mention in Section 2. We also analyzed MobileMedia [13], a PL for media (photo, video and audio) management on mobile devices. MobileMedia has been refactored and evolved to incorporate new features in order to address new scenarios and applications along many releases. It has been implemented in two different versions: (i) an object-oriented Java implementation that uses conditional compilation to implement variabilities; and (ii) an aspect-oriented Java implementation that uses AspectJ aspects to modularize the PL variabilities, which we used in this study. One of the authors was involved in TaRGeT development and one other author worked as a developer in MobileMedia AspectJ implementation.

We divided the analysis of TaRGeT in two different steps. First, we analyzed TaRGeT evolution from release 4.0 to release 5.0 (from January to July 2009) to derive the transformation templates that we presented in Section 4. Then, in the second step, we analyzed TaRGeT evolution between releases 5.0 and 6.0 (from July 2009 to January 2010) in order to verify if the templates that we had previously identified could address the changes performed during this period. As mentioned in Section 2, we identified 11 safe evolution scenarios between releases 4.0 and 5.0 and 9 safe evolution scenarios between releases 5.0 and 6.0. When analyzing MobileMedia, we evaluated its evolution between releases 4 to 8 and we identified 8 safe evolution scenarios performed in the PL during this period.

For both PLs, we had the code of all releases and the evolution history available in SVN repositories. To identify the evolution scenarios, we first compared the FM and CK of each release to verify which features and assets were added, removed or modified from one release to another. Each of these operations (feature addition, removal or modification) corresponded to an evolution scenario. We then analyzed the scenarios and we discarded the ones that were not safe evolutions according to the PL refinement notion, for example, introduction of a mandatory feature or bug correction. Regarding the scenarios that involved assets modification, we also inspected the assets history in the SVN repository, comparing different versions of each of them to discover how the developers implemented the changes. For this, we also relied on commit comments and revision history annotations present in source files that described the changes that developers executed.

[1] `http://twiki.cin.ufpe.br/twiki/bin/view/SPG/SPLRefactoringTemplates`

5.2 Interpretation

In our study, we also examined whether our templates were sufficient to express the modifications implemented in the identified safe evolution scenarios. Table 2 lists the frequency of use for each template in TaRGeT and MobileMedia PLs. For each safe evolution scenario that we analyzed, we listed which templates would be necessary to address the modifications performed by the developers. For example, if the implementation required extracting parts of a class to an aspect, it would be necessary to apply the Split Asset template several times until all code was moved to the aspect. In these cases we only list the template once, even if it is used more than one time in the same scenario. This is because we want to verify if our set of templates was sufficient to express the safe evolution scenarios implementation, no matter how many times they were used.

According to Table 2, we observed that in the TaRGeT PL, Template 6 - Add New Alternative Feature was the most widely used in the analyzed safe evolution scenarios. We believe that this was due to the implementation of different formats for input use case documents (Word, XML, XLS) and output test suites (XML, HTML and XLS). On the other hand, we did not find any occurrence of Template 8 - Delete Asset.

Among MobileMedia safe evolution scenarios, we did not found any occurrence of Template 7 - Add New OR Feature because it does not have any OR feature. Template 1 - Split Asset and Template 2 - Refine Asset were the most used templates. The former was applied with code templates to extract subclass and to extract class member to aspect. The latter was applied in release 5 to introduce a new alternative feature in the PL, which was also relied on Template 6 - Add New Alternative Feature. Finally, Template 8 - Delete Asset was used once when an exception handling class became no longer useful.

Another fact that we have noticed when analyzing both PL history was that some operations usually involved a large number of modified classes. In TaRGeT PL, the average number of modified assets was 12.9 and in MobileMedia PL the average was 11 assets. This might indicate that if we implement the templates together with a development tool, we could also improve productivity. However, evaluate this is not the focus of our work.

In both cases we found that our transformation templates, in addition to the FM refactorings listed in previous work [1] were sufficient to address the modifications performed in the analyzed safe evolution scenarios, which reinforces the expressivity of our safe evolution templates. Besides, during our study, we identified 2 cases that actually introduced new bugs in the PL, in the second step of TaRGeT's analysis. We described one of these errors in Section 2. The other error was similar to this one and involved the incorrect association of a feature expression in the CK for assets related to a new alternative feature. We can avoid this error using Template 6 - Add new alternative feature.

In MobileMedia we found 4 cases that introduced errors in the PL. One of these cases involved extraction of class fields to an aspect that was associated to an incorrect feature expression in the CK. This problem could be avoided by using Template 1 - Split Asset. The other errors are about incorrect association of feature expressions in the CK, that could be avoided using Template 5 - Replace Feature Expression.

We also have proved soundness for all templates listed in Table 1 using the Prototype and Verification System (PVS) [20]. This formal proof is done in accordance with the general PL refinement

Table 2. Templates Frequence

Template	TaRGeT	MobileMedia
Split Asset	4 (13.79%)	6 (25%)
Refine Asset	5 (17.24%)	6 (25%)
Add New Optional Feature	5 (17.24%)	4 (16.66%)
Add New Mandatory Feature	3 (10.34%)	1 (4.16%)
Replace Feature Expression	3 (10.34%)	4 (16.66%)
Add New Alternative Feature	8 (27.58%)	2 (8,33%)
Add New OR Feature	1 (3.44%)	0 (0%)
Delete Asset	0 (0%)	1 (4,16%)

theory (See Section 3). The templates proof details are not in the scope of this work, but they can be retrieved in our website.[2]

It is important to mention that we only guarantee that the templates' application does not introduce errors in the PL if the developer follows exactly the steps described in the transformations, respecting the restrictions. We acknowledge that it would be ideal to have the templates semi-automated together with development tools in order to avoid possible mistakes developers might make when applying the templates manually. However, the templates are an useful guide to help developers to perform safe modifications in the PL.

These two aspects are important because together they address the problem of the likely chance of error in manual PL evolution; we assure that the templates do not introduce new bugs and also that the developer does not need to perform modifications that are not described by the templates.

5.3 Threats to Validity

Since we performed our analysis manually and we only verified the PL changes between the available releases, it is possible that some evolution scenarios have not been taken into consideration. In this case we may have missed evolution steps that could not be justified by our templates or scenarios that would enhance the expressivity of our set of templates. However, we believe that the evolution scenarios that we did not analyze are mostly related to code refactorings and we already have a template to deal with this kind of change, namely Template 2 – Refine Asset. We could missed some code transformation templates by not analyzing these refactorings, but we know that there are a great number of code templates that can be used with Template 2 and it is not our intention to list them all in this work.

Our approach is based on the fact that manually evolving PLs is error-prone. In this study we found evidences of this issue by identifying evolution scenarios that were supposed to preserve the behavior of existing products but in fact introduced errors in the PL. Despite this, it is also possible that we have overlooked errors introduced by the manual changes during PL evolution and that the number of errors is even bigger, which encourages us to continue with our research.

Another threat to our work is the fact that MobileMedia is a small system that was developed for educational purposes. However, we observed that the same categories of evolution scenarios that we identified in TaRGeT PL were present in MobileMedia, which indicates that these scenarios are relevant to our study.

We believe that the fact that we already knew the available templates when we were analyzing the two PLs evolution did not influenced our results, because first we listed the modifications the developers executed and then we tried to match these changes with the available templates. If none of the templates could be applied, we analyzed if it was necessary to modify some of them or to create a new template that could address the changes.

[2] http://twiki.cin.ufpe.br/twiki/bin/view/SPG/TheorySPLRefinement

Because of the limited quantity of PLs analyzed, the quantitative results cannot be generalized with confidence. However, the qualitative results are an evidence that our set of safe evolution templates is quite expressive.

6. Related Work

The notion of PL refinement discussed here first appeared in a PL refactoring tutorial [4]. Besides covering PL and population refactoring, that tutorial illustrates different kinds of refactoring transformation templates that can be useful for deriving and evolving PLs. Another work [5] extended this initial formalization making clear the interfaces between the PL refinement theory and languages used to describe PL artifacts. In our work we use the existing definitions for PL refinement and the idea of safe evolution transformation templates, but we go further by proposing other new templates based on the analysis of a real PL and evaluating these templates in two other PLs.

Alves et al. [1] proposed a refactoring notion for PL relating FMs and described a catalog of FM refactorings. Our work is complementary to this one because the PL refinement notion that we rely on, and consequently our templates, supports other PL artifacts like CK and assets, in addition to FM.

Early work [9] on PL refactoring focus on Product Line Architectures (PLAs) described in terms of high-level components and connectors. This work presents metrics for diagnosing structural problems in a PLA, and introduces a set of architectural refactorings that can be used to resolve these problems. Besides being specific to architectural assets, this work does not deal with other PL artifacts such as FMs and CK, as do the safe evolution templates presented in our work. There is also no notion of behavior preservation for PLs.

Several approaches [16, 17, 19, 24] focus on refactoring a product into a PL, not exploring PL evolution in general, as we do here with our templates. First, Kolb et al. [17] 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 PL assets. There is no discussion about FMs and CK. Similarly, Kastner et al. [16] focus only on transforming code assets, implicitly relying on refinement notions for aspect-oriented programs [8]. As discussed here and elsewhere [4] these are not adequate for justifying PL refinement. Trujillo et al. [24] go beyond code assets, but do not explicitly consider transformations to FM and CK as our templates do. 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. [19] 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 the refinement notion that we rely on, 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 their technique can be mapped to the notion of asset refinement, both theories could be used together.

Thüm et al. [23] present and evaluate an algorithm to classify edits on FMs. They classify the edits in four categories: refactorings, when no new products are added and no existing products are removed; specialization, meaning that some existing products are removed and no new products are added; generalization, when new products are added and no existing products removed and arbitrary

edits otherwise. In our work, we also analyzed edits in other artifacts like CK and code assets, in addition to FM. However, we are only interested in refactorings and generalization edits, not considering specialization and arbitrary edits.

7. Conclusions

In this paper we investigate the safe evolution of product lines and based on the results of this study we present and describe a set of safe evolution templates that can be used by developers in charge of maintaining product lines. The described templates abstract, generalize and factorize the analyzed scenarios and are in accordance with the refinement notion that we rely on. The templates express transformations in feature models and configuration knowledge. We abstract code assets modifications through code transformation templates, which are more precise transformations that deal with changes in code level. Some of our general product line templates might have a set of code transformation templates associated to them.

We also present the preliminary results of a study that we performed to evaluate the evolution of two product lines. We show evidence that the discovered templates can address the modifications performed in the safe evolution scenarios that we identified analyzing the SVN history of these two product lines. We present examples of how using our templates could avoid the mistakes that we found during our analysis and we show the frequency of occurrence of each template among the analyzed scenarios. These results, in addition to the templates formal proofs, intend to address the problem of the likely chance of errors in manual evolution in product lines.

We know that our results are limited by the context of the two product lines that we analyzed and that new templates (both general and specific for code assets) can be necessary to justify other transformations that we did not analyze in this paper. In order to complement these results, we intend to extend our study by analyzing other product lines from different domains.

Our results also show evidence that product line manual evolution can be time consuming, because it usually involves the analysis and modification of a great number of source code artifacts. We believe that the templates automation integrated with a development tool could address this issue. As future work, we intend to implement our templates integrated with FLiP [3], an existing product line refactoring tool developed by our group. We also plan to execute a controlled experiment to evaluate the time, and consequently, productivity gains when using our templates to evolve product lines.

Acknowledgments

We would like to thank CNPq, a Brazilian research funding agency, and National Institute of Science and Technology for Software Engineering (INES), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08, for partially supporting this work. Laís is supported by CNPq, grant 131499/2010-6. We also thank the anonymous reviewers for their detailed comments. In addition, we thank SPG[3] members for feedback and fruitful discussions about this paper.

References

[1] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. J. P. de Lucena. Refactoring product lines. In *GPCE 2006, Portland, Oregon, USA*, pages 201–210. ACM, 2006.

[2] V. Alves, P. Matos, L. Cole, A. Vasconcelos, P. Borba, and G. Ramalho. Extracting and evolving code in product lines with aspect-oriented programming. *Transactions on Aspect-Oriented Software Development*, 4:117–142, 2007.

[3] V. Alves, F. Calheiros, V. Nepomuceno, A. Menezes, S. Soares, and P. Borba. Flip: Managing software product line extraction and reaction with aspects. In *SPLC*, page 354, 2008.

[4] P. Borba. An introduction to software product line refactoring. In *GTTSE'09 Summer School*, Braga, Portugal, 2009.

[5] P. Borba, L. Teixeira, and R. Gheyi. A theory of software product line refinement. In *ICTAC'10*, pages 15–43, Berlin, Heidelberg, 2010. Springer-Verlag.

[6] A. Cavalcanti, P. Borba, A. Sampaio, and M. Cornelio. Algebraic reasoning for object-oriented programming. *Science of Computer Programming*, Jan. 2004.

[7] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[8] L. Cole and P. Borba. Deriving refactorings for AspectJ. In *AOSD'05*, pages 123–134. ACM Press, 2005.

[9] M. Critchlow, K. Dodd, J. Chou, and A. van der Hoek. Refactoring product line architectures. In *1st International Workshop on Refactoring: Achievements, Challenges, and Effects*, pages 23–26, 2003.

[10] K. Czarnecki and U. Eisenecker. *Generative programming: methods, tools, and applications*. Addison-Wesley, 2000.

[11] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *GPCE 2006*, pages 211–220, 2006.

[12] F. Ferreira, L. Neves, M. Silva, and P. Borba. Target: a model based product line testing tool. In *Tools Session of CBSoft 2010*, Salvador, Brazil, 2010.

[13] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. C. Ferrari, S. S. Khan, F. C. Filho, and F. Dantas. Evolving software product lines with aspects: an empirical study on design stability. In *ICSE*, pages 261–270. ACM, 2008.

[14] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Aug. 1999.

[15] K. Kang, S. Cohen, J. Hess, W. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, SEI, CMU, 1990.

[16] C. Kastner, S. Apel, and D. Batory. A case study implementing features using AspectJ. In *SPLC*, pages 223–232, 2007.

[17] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. A case study in refactoring a legacy component for reuse in a product line. In *21st ICSM*, pages 369–378. IEEE Computer Society, 2005.

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

[19] J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *ICSE'06*, pages 112–121. ACM, 2006.

[20] S. Owre, J. Rushby, and N. Shankar. Pvs: A prototype verification system. In *11th International Conference on Automated Deduction*, pages 748–752. Springer-Verlag, 1992. ISBN 3-540-55602-8.

[21] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[22] D. B. Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois, 1999.

[23] T. Thüm, D. S. Batory, and C. Kästner. Reasoning about edits to feature models. In *ICSE*, pages 254–264. IEEE, 2009. ISBN 978-1-4244-3452-7.

[24] S. Trujillo, D. Batory, and O. Diaz. Feature refactoring a multi-representation program into a product line. In *GPCE'06*, pages 191–200. ACM, 2006.

[25] F. van der Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action: the Best Industrial Practice in Product Line Engineering*. Springer, 2007.

[3] http://www.cin.ufpe.br/spg