The Journal of Systems and Software 106 (2015) 42–58

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

Safe evolution templates for software product lines

L. Neves [a], P. Borba [a], V. Alves [b], L. Turnes [b], L. Teixeira [a,*], D. Sena [c], U. Kulesza [c]

[a] Informatics Center, Federal University of Pernambuco, Av. Jornalista Anibal Fernandes, s/n 50740–560 Recife, PE, Brazil
[b] Computer Science Department, University of Brasilia, Brazil
[c] Department of Informatics and Applied Mathematics, Federal University of Rio Grande do Norte, Natal, Brazil

ARTICLE INFO

Article history:
Received 11 February 2014
Revised 3 March 2015
Accepted 3 April 2015
Available online 16 April 2015

Keywords:
Software product lines
Refinement
Evolution

ABSTRACT

Software product lines enable generating related software products from reusable assets. Adopting a product line strategy can bring significant quality and productivity improvements. However, evolving a product line can be risky, since it might impact many products. When introducing new features or improving its design, it is important to make sure that the behavior of existing products is not affected. To ensure that, one usually has to analyze different types of artifacts, an activity that can lead to errors. To address this issue, in this work we discover and analyze concrete evolution scenarios from five different product lines. We discover a total of 13 safe evolution templates, which are generic transformations that developers can apply when evolving compositional and annotative product lines, with the goal of preserving the behavior of existing products. We also evaluate the templates by analyzing the evolution history of these product lines. In this evaluation, we observe that the templates can address the modifications that developers performed in the analyzed scenarios, which corroborates the expressiveness of our template set. We also observe that the templates could also have helped to avoid the errors that we identified during our analysis.

1. Introduction

A software product line is a set of related software products that are systematically generated from reusable assets. Products are related in the sense that they share functionality or behavior. Assets correspond to artifacts such as classes and property files, which we compose or instantiate in different ways to specify or build products. This kind of reuse targeted at a specific set of products can bring productivity and time-to-market improvements (van der Linden et al., 2007; Pohl et al., 2005). To obtain these benefits with reduced upfront investment and risks, we can minimize the initial upfront cost of software product line adoption by extracting a software product line from existing products (Clements & Northrop, 2001; Krueger, 2002). Similar processes apply to evolving a software product line, both when just improving the product line design and when adding new functionality and products, which often requires extracting variations from parts previously shared by a set of products.

The activity of manually extracting different software product line assets when evolving it requires substantial effort, especially for checking necessary conditions to make sure the extraction is correctly performed. In fact, the lack of specific guidelines and development tools to support software product line evolution makes this process error-prone. Extractions might lead to unintended modifications to the behavior of existing products, affecting product line users and compromising the promised benefits in other dimensions of costs and risks. The associated defects are more difficult to track because they are only present in specific products. Generating and testing these different products may help to discover and correct the mentioned issues. However, as the number of products can be high, testing all of them can be expensive and impact productivity.

To avoid these problems and evolve a software product line in a *safe* way (in the general sense that behavior is preserved, not specifically referring to conventional safety properties), we could resort to a formal notion of software product line refinement or safe evolution (Borba, 2009; Borba et al., 2012). By basically requiring preservation of the observable behavior of existing products, this notion guarantees that changes to a software product line do not impact its existing users. So users of the products that could be generated before the changes can use the new modified products without noticing any difference. This notion applies when we need to introduce new products to the software product line without changing existing ones, or when we want to improve the product line design without modifying the behavior of existing products. To support such change scenarios, safe evolution considers that software product line specific artifacts, like feature models (Czarnecki & Eisenecker, 2000; Kang et al., 1990) and configuration knowledge (Czarnecki & Eisenecker, 2000), often

* Corresponding author.
E-mail addresses: lmn3@cin.ufpe.br (L. Neves), phmb@cin.ufpe.br (P. Borba), valves@unb.br (V. Alves), lmt@cin.ufpe.br (L. Teixeira), demostenes.sena@ifrn.edu.br (D. Sena), uira@dimap.ufrn.br (U. Kulesza).

http://dx.doi.org/10.1016/j.jss.2015.04.024
0164-1212/

coevolve with assets (Neves et al., 2011; Passos et al., 2013; Seidl et al., 2012).

With the goal of better understanding the process involved in software product line safe evolution, in this work we describe empirical studies that lead to the discovery and analysis of 67 concrete safe evolution scenarios from five different software product lines. Each scenario is characterized by a commit and one of its subsequent commits in the evolution history of the product lines repositories.

Based on the evolution history from two of the aforementioned software product lines, we identify and precisely describe a number of safe evolution templates that abstract, generalize, and factorize the analyzed scenarios, and also conform to the refinement notion (Borba et al., 2012) we rely on. These templates are generic transformations that developers can safely apply when maintaining compositional and annotative software product lines. They specify transformations that go beyond program refactoring notions (Fowler, 1999; Roberts, 1999), which deal with simple programs, by considering both sets of reusable assets that do not necessarily correspond to valid programs, and extra software product line artifacts such as feature models and configuration knowledge. For each template, we describe its structure and the necessary conditions for proper application. We also show examples of correct application of the templates in evolution scenarios mined from existing software product lines. This way we hope to provide extra, concise and explicit guidance to evolve a software product line in a safe way. The templates can also be used as a basis to automate support for safe software product line evolution.

We evaluate the proposed templates by analyzing the evolution history of the five aforementioned software product lines. In this evaluation, we could observe that the proposed templates can address the modifications that developers performed in the analyzed scenarios, which corroborate the expressiveness of our template set. As a secondary result, we observe that the templates could be used to avoid some defects introduced during the evolution history of some product lines. Such defects were caused by modifications that were supposed to be safe, but actually changed the behavior of existing products.

In summary, with the aim of discovering more safe evolution templates and assessing whether they could be useful to justify existing evolution scenarios, this article extends our previous conference paper (Neves et al., 2011) in two main ways:

- study of annotative software product lines: we go beyond our previous study on compositional software product lines by analyzing and presenting five additional templates (Section 3.2.2) that deal with an extended configuration knowledge notion—mapping feature expressions to transformations involving assets—necessary to address evolution scenarios that involve preprocessor-based variability management in annotative software product lines (Kästner et al., 2008);
- further evaluation: we bring additional evidence of the expressiveness of the proposed templates, evaluating the evolution history of three additional software product lines, namely, a product line of research group management systems, a product line of product line derivation tools, and a product line of academic information systems.

We organize the rest of the text as follows. Section 2 introduces the main concepts used in this work, such as feature models, configuration knowledge, asset mappings, and the product line refinement notion. Section 3 presents the safe evolution templates for software product lines. It also shows some examples of the templates being applied in different software product lines. Section 4 presents the results of a study performed to evaluate the expressiveness of our template set. We discuss related work in Section 5 and conclude with Section 6.

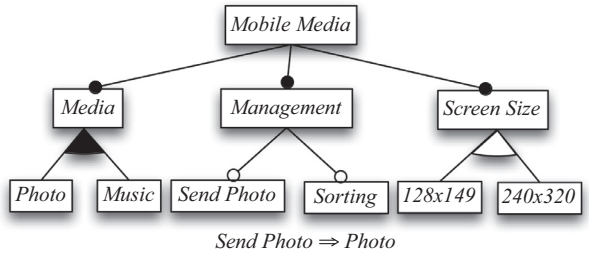


Fig. 1. MobileMedia FM example.

2. Software product line concepts

To enable the automatic generation of products from assets, software product lines, hereafter product lines, often rely on artifacts such as Feature Models (FM), Configuration Knowledge (CK) (Czarnecki & Eisenecker, 2000), and assets, which we briefly describe in what follows. To guide the product line evolution analysis and identify the evolution scenarios, we rely on a product line refinement notion (Borba et al., 2012), which formalizes our intuition about safe evolution, drives the analysis of the evolution scenarios and justifies the product line transformation templates we propose in this article. Essentially, we say that a product line L' refines a product line L whenever L' is able to generate products that behaviorally match L products. This way, users of a product from L cannot observe behavioral differences when using the corresponding product of L'. This is exactly what guarantees safety when improving a product line design by changing its FM, CK or assets.

2.1. Feature models

A FM is usually represented as a tree, containing features and information about how they relate to each other. Features have different names and abstract groups of associated requirements, both functional and non-functional. In this work, we use the notation by Czarnecki and Eisenecker (2000) to express relationships between a parent feature and its child features.

Besides these relationships, the notation we consider may also contain propositional logic constraints over features. We use feature names as atoms to indicate feature selection. So, negation indicates that a feature should not be selected. For instance, the formula below the tree in Fig. 1 states that feature *Photo* must be present in every product that has feature *SendPhoto*. So {*Photo, SendPhoto*, 240×320}, together with the mandatory features, which hereafter we omit for brevity, is a valid feature selection (product configuration), but {*Music, SendPhoto*, 240×320} is not. A product configuration is a valid feature selection that satisfies all FM constraints, specified both graphically and through formulae. Each product configuration corresponds to a product from the product line, expressed in terms of the features it supports. This captures the intuition that the FM denotes the set of products in a product line (Schobbens et al., 2007).

2.2. Assets

In a product line, we specify and implement features with reusable assets. So, we must consider different languages for specifying and implementing assets such as requirements documents, design models, code, tests, data files, and so on. For simplicity, in the text we focus on code assets for the examples and concepts, as they are equivalent to the other kinds of assets with respect to our interests on product line refinement. This way, we can focus on the essential concepts these languages should support. The important issue here is not the

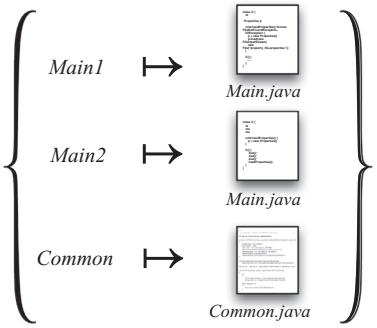


Fig. 2. MobileMedia asset mapping example.

nature of the assets contents, but actually how they are compared and referred to in a product line.

First, we assume the concept of well-formedness of asset sets, captured by a *wf* predicate over asset sets. In a number of programming languages, this corresponds to the notion of a well-typed program, but it may also correspond to purely syntactical or less strict semantic constraints. For most product lines, *wf* should be instantiated by a function that considers the well-formedness of sets of assets written in different specification and implementation languages.

Second, we rely on means of comparing assets with respect to behavior preservation. This may range from lexical relations that basically compare assets for textual equality to semantic relations based on testing or formal refinement of asset sets. In particular, we assume the relation $\sqsubseteq$ denotes refinement of an asset set. This set might correspond to a full product, or even a single asset, depending on the specifics of a refinement notion used for a particular language. To support stepwise refinement and evolution, we require that the refinement notion must be a pre-order, as is the case of existing notions for object-oriented programming languages, such as ROOL (Refinement Object-Oriented Language), which is based on the sequential subset of Java (Borba et al., 2004). Finally, asset set refinement must be compositional in the sense that refining a set of assets that is part of a well-formed product yields a refined well-formed product. Such a compositionality property is a common property of refinement relations, such as the one described in Borba et al. (2004), and it is essential to guarantee independent development of assets in a product line. It is useful when applying transformations that change the contents of some assets, such as in the SPLIT ASSET template. Other templates, such as ADD NEW OPTIONAL FEATURE, can be proved even without such property.

These are the concepts we need to specify our templates, which actually apply to any combination of asset languages having such notions of well-formedness and behavior preservation between asset sets. However, to enable unambiguous references to assets, instead of considering that a product line contains simply a set of assets, we assume that it contains a mapping as illustrated in Fig. 2. This mapping associates asset names referenced in other parts of the product line specification to actual assets. Such an asset mapping basically corresponds to an environment of asset declarations. This allows conflicting assets in a product line, like assets that implement alternative features, such as both *Main* classes in the illustrated asset mapping.

2.3. Configuration knowledge

As discussed in Section 2.1, features group requirements, so they must be related to the assets that realize them. The CK specifies this relation. We can express the CK with varying degrees of expressiveness (Kästner et al., 2008). Based on the product lines we analyzed, here we consider two different CK notations. We first explain a simpler notation used by the compositional product lines studied. Later we explain an extended notation used by the annotative product lines, which use preprocessing as the main variability implementation mechanism.

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

Fig. 3. Part of the MobileMedia compositional CK.

RGMS	*Member, Publication*
Database	*HibernateConfig*
ResearchLine	*preprocess Member tag researchLine*
⋮	⋮

Fig. 4. RGMS transformation CK example.

In compositional product lines, we compose assets to implement features. So we basically need a CK notation that maps feature expressions, denoting presence conditions (Czarnecki & Pietroszek, 2006), to sets of asset names. Propositional formulae having feature names as atoms represent these feature expressions. For example, the MobileMedia CK illustrated in Fig. 3 establishes that if the *Photo* and *Music* features are both selected, then the *AppMenu* aspect (Kiczales et al., 2001), among other assets omitted in the fifth row, should be part of the final product.

Essentially, this product line uses aspects as a variability implementation mechanism (Alves et al., 2007; Gacek & Anastasopoules, 2001). In particular, the *AppMenu* aspect, due to usability issues, should not be present in products that have only one of the Media features, which is what the fifth row specifies. Given a product configuration, CK evaluation yields the assets that constitute the corresponding product. In this example, the configuration {*Photo, 240* × *320*} leads to the actual assets associated with the following names: {*MM.java*, . . . , *Photo.java*, . . . , *Common.aj*, . . . }. This captures the semantics of CK evaluation.

In the context of annotative product lines, we have fine-grained variability specified in terms of preprocessing annotations within assets. Therefore, asset selection as in compositional product lines (Neves et al., 2011) is not enough, since we have to process assets to generate products. So we need a more elaborate CK notation, that maps feature expressions into transformations dealing with assets.

To illustrate this notation, we show in Fig. 4 a simplified CK of the Research Group Management System (RGMS) product line. We can select assets, as in the first two rows. This is actually seen as a simple transformation, *select and move components*, which selects the assets listed in the CK and includes them in the generated products if their corresponding feature expression evaluates to true. For simplicity, and compatibility with the previous notation, we omit the transformation name. On the third row we can see more elaborate transformations, such as *preprocess* and *tag*. The first is responsible for preprocessing assets that contain conditional compilation directives. It receives as parameter the asset names to be preprocessed. In the example, we specify that the *Member* asset should be preprocessed when we select the *ResearchLine* feature. The other transformation in the same row specifies that the value of the *researchLine* conditional compilation tag should be set to true when the associated feature

expression evaluates to true. So, by selecting the *ResearchLine* feature, we trigger the preprocessing of the *Member* asset in a context where the *researchLine* tag is set to true. As a consequence, *Member* code delimited by preprocessing directives such as *#if($researchLine)* and *#end* are included in the version of *Member* used to build products with the *ResearchLine* feature.

Given a product configuration, CK evaluation proceeds by evaluating feature expressions and later applying the transformations associated to the valid expressions. The interpreter first considers the *select and move components* transformations, followed by the *tag* transformations, and then the *preprocess* transformations. Tags not set by applying *tag* transformations are disabled when preprocessing assets. Assets not associated to a *preprocess* transformation are not preprocessed, even if they are selected and contain preprocessing directives. This likely leads to non well-formed products, and consequently a non well-formed product line, as we discuss later. To avoid that, in both presented CK notations, similar care should be taken to refer to existing names of features in the FM and valid names of assets in the asset mapping. Further, to assure that the CK is well-formed, we have to use valid transformation names and properly provide arguments to these transformations.

3. Safe evolution templates

The product line refinement notion (Borba et al., 2012) supports us on safely evolving a product line, by improving its design or extending it with new products while preserving existing ones. However, it may be difficult to precisely reason about the definitions when evolving an product line. Minor imprecisions often lead to the oversight of issues that compromise the safety of the evolution process. On the other hand, evolving a product line in an *ad hoc* way is error-prone because, to make sure that the behavior of existing products is preserved, one usually has to analyze different interdependent artifacts, like FMs and CK, in addition to the product line assets. Another difficulty is that the defects that one might introduce during product line evolution can be difficult to detect because they are present only in certain products, and the configuration space might be large. Generating and testing all these different products may help to discover and fix problems like the one we just described, but, as the number of generated products can be high, testing all of them can be expensive and impact productivity. Therefore, supporting safe evolution tasks by means of templates might avoid such risks and costs.

To avoid these problems and provide guidance to developers, in this section we propose a number of product line safe evolution templates. Each template specifies a generic product line transformation that captures modifications to the product line elements (FM, CK, and asset mapping) in accordance to the product line refinement notion. This way, by applying a template to a specific product line, we obtain an improved product line that refines the former and therefore is able to generate products that preserve the behavior of products of the original product line.

3.1. Templates discovery

To discover the safe evolution templates that might help to avoid the issues just discussed, we analyzed concrete evolution scenarios from two product lines mentioned earlier: TaRGeT and Research Group Management System (RGMS). A scenario consists of a commit and one of its previous or subsequent commits in the evolution history of the product line repositories. Based on such analysis, we identified which scenarios corresponded to refinements, and then formalized and proved (detailed elsewhere (Borba et al., 2012)) the templates according to the product line refinement theory. In what follows, we detail the template discovery process.

To discover refinement templates that work for both compositional and annotative product lines, we used TaRGeT, a product line of tools that generate functional tests from use case specifications. We considered evolution scenarios from releases 4.0[1] (10 features and 20 KLOC) to 5.0 (27 features and 40 KLOC).

To derive templates for annotative product lines, we analyzed RGMS, a product line of systems for managing research groups. RGMS was initially developed as part of a graduate course on product lines in which we had 5 pairs of students working as developers of their own RGMS implementation. The average number of implemented features in the five implementations is 20, with approximately 3 KLOC. During the course, each pair developed 3 releases. So, we had 15 safe evolution scenarios available in total, from which we used 2 to discover templates. We chose one scenario from release 1 and another from release 2 of the same implementation, because the tasks performed in these releases involved the modification of many assets, and therefore had the potential to be a richer source for mining templates.

We followed an iterative and incremental process for discovering templates. First, we examined the evolution history of these product lines, looking for evolution scenarios that were supposed to be safe according to the refinement notion. For each potentially safe evolution scenario, we manually inspected the involved artifacts to check whether the result of the evolution step in fact amounted to changes that preserved the behavior of existing products. This involved reading the code and understanding the changes, and also running some products from the product line. We also compared the FM, CK, and AM to identify which features and assets were added, removed or modified in each step. Assets included Java and AspectJ code, as well as Eclipse plugin configuration files, image files, and so on. Each operation (addition, removal, or modification) corresponded to an evolution scenario. In addition to FM and CK changes, scenarios also involved asset modification. Thus, we often had to inspect the asset evolution history in the source code repository. For example, to discover the changes made to a Java class, we manually inspected different commits of this class using the version control system of the product line. We also considered commit messages and revision history annotations in the source files. Therefore, we often had to analyze more than one commit to understand a change. Finally, in some cases, we also talked to the developers involved in the change, to further understand the rationale behind it.

We then identified the modifications involved in each scenario and tried to discover patterns in the changes developers performed. For example, we observed that when making a mandatory feature optional developers often had to track the code related to the feature and then extract it to a different asset. The scenarios that passed the inspection were then separated into groups of scenarios that involved similar kinds of changes to the product line artifacts, such as adding a feature, or splitting an asset. This process was manually performed, informally considering the refinement notion when comparing different versions of the product line artifacts. Based on these groups, we selected scenarios that could be considered safe evolutions according to the product line refinement notion, to mine templates that could generalize the observed situations. We analyzed each group to identify commonalities and abstract the differences, proposing a refinement template.

Finally, in some occasions, we also factored out some coarse-grained scenarios into many fine-grained changes, which resulted in more than one refinement template created. Such scenarios were better abstracted by the application of different templates in sequence, rather than by a single very specific template that performed many changes. We could do this due to the transitivity of the product line refinement notion. Since each template is proved as a product line refinement, this allows us to divide a template in two and sequentially apply them. We only have to assure that the resulting

[1] Prior to release 4.0, TaRGeT was not a product line.

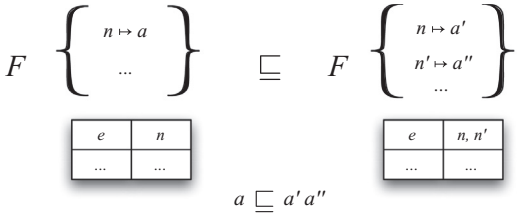


Fig. 5. Split asset.

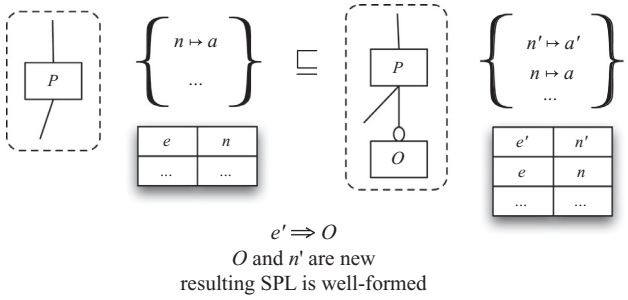


Fig. 6. Add new optional feature.

product line of the first template matches with the original product line of the second template.

3.2. Safe evolution templates

As a result of the discovery process just described, we present a set of safe evolution templates, which specify changes to product line artifacts in accordance to the product line refinement notion. By complying with the refinement notion, we ensure that changes associated to template application preserve the behavior of existing products.

Although the product line refinement notion we rely on is independent of the languages one employs to specify these artifacts, the safe evolution templates describe transformations using specific notations for FM and CK, in such a way that we can detail how these artifacts should be changed. Most templates are also independent of the language we use to specify or implement assets. Some assume a specific notation for preprocessing directives, that could be adapted to similar notations. So, in effect, we do not lose generality and are still able to precisely specify the associated changes.

Besides specifying the allowed changes to FM, CK and assets, each template describes preconditions that must be satisfied in order to guarantee that the transformation is indeed safe. We illustrate this next, focusing first on templates that are useful for both compositional and annotative product lines. Then we focus on templates for annotative product lines; they refer explicitly to either preprocessing directives in the involved assets or preprocessing transformations of the more elaborate CK notation. Note that, as a consequence of the factorization step of the template discovery process, each template focuses on small changes to a product line. Nevertheless, we can derive more elaborate refinement transformations by composing the ones proposed here.

3.2.1. Templates for compositional and annotative product lines

After analyzing part of TaRGeT's evolution history, we discovered a set of eight templates, a subset of which we describe in this section. The description of the complete set and more details about this work are available in our online appendix.[2] As these templates do not explicitly refer to preprocessing mechanisms and associated CK transformations, they can be used for both compositional and annotative product lines.

Split asset. When analyzing evolution scenarios that changed a mandatory feature to optional, we observed that this type of evolution usually first involved tracking the code related to the feature and extracting it to other artifacts, such as other classes and aspects, for instance. To generalize the cases where a part of an asset that is related to a specific feature is extracted to another asset, we derived Split asset, illustrated by Fig. 5. This template specifies when a given asset can be split into two, and how the CK must be correspondingly updated to preserve the behavior of products that include the original asset.

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

In a template, we specify the changes to the FM, CK and asset mapping using meta-variables to abstractly represent the elements. In this case, we do not change the FM, so we have F on the source (left of the refinement symbol $\sqsubseteq$) and target (right of the same symbol) product lines. Feature expression e and asset name n also appear in the CK of both product lines. For asset mappings and CK, we use the meta-variables A and K, but these do not appear in the template, since it details their structure. The CK is a table with meta-variables. Those in the left represent feature expressions, and those in the right represent asset names. Ellipsis indicate that the CK can have other rows besides the ones detailed. This template specifies that we change the CK by associating a new asset name n' to the expression e. The asset mapping groups associations of asset names to assets. In this template, we specify that the original mapping is changed by adding a new association (from n' to a'') and replacing a for a'.

The precondition at the bottom of Fig. 5 specifies that we can only split an asset a into two other assets a' and a'' when the new assets $a'a''$ refine asset a. This restriction is necessary to guarantee that the behavior of source products that contain a is matched by target products that actually contain $a'a''$. Thus, we leverage the notion of asset refinement from each asset language. This way, the template could be applied to product lines formed by assets written in any language with a proper refinement notion, as formalized elsewhere (Borba et al., 2012).

Another restriction establishes that n' must be a new asset name. Without this restriction we could have an existing asset incorrectly associated to e. The last restriction helps us to simplify matters by assuming a canonical representation for the CK. Otherwise, we would have to express the fact that all occurrences of n in the CK should be changed for n, n'. This allows us to separate concerns and abstract this possibility; other templates could be used for moving, splitting and merging CK items (Borba, 2009). Based on these conditions, we can say that Split asset assures product line refinement because, for each product that contained asset a in the source product line, there is now a corresponding refined one in the target product line that contains the composition of assets a' and a''. We can prove this through compositionality of the asset refinement notion (Borba et al., 2012), which also ensures that the resulting product line is well-formed.

Add new optional feature. By analyzing evolution scenarios that consist of adding new features to the product line, we derived Add new optional feature. It is useful when a developer needs to introduce an optional feature without changing existing assets, as Fig. 6 presents. Notice that in this case, the set of possible products is augmented. The template states that we can introduce a new optional feature O together with a new asset a', as long as this asset is only included in products that have O. In this template, we detail the FM showing the meta-variables that explain adding the new feature. We represent the fact that a feature may have other features related to it using a line above or below it. Additionally, we detail the first row in the source CK, which associates e with n, to illustrate that the existing CK items remain unchanged after the transformation.

The precondition states that we cannot have another feature named *O* in the FM, nor another asset name *n'* in the asset mapping, as this would lead to invalid artifacts. The template also requires the resulting product line to be well-formed, to guarantee that the new asset a' properly composes with other assets in the new products. This avoids assets with conflicting names in a product, since in most languages this causes compilation errors. We can check the well-formedness restriction using the safe composition approach (Teixeira et al., 2013; Thaker et al., 2007). In some templates, we do not explicitly require this because we can deduce it from the template and precondition constraints. In this template, the resulting product line has the same products that it had before, in addition to products that contain the new *O* feature.

The well-formedness restriction also implicitly deals with the problem of feature interactions (Kaestner et al., 2009). Introducing a new feature can impact others. However, in ADD NEW OPTIONAL FEATURE, the new feature does not impact existing products, since any asset associated with it is only present when we select this feature. The existing products do not contain such feature, and thus remain syntactically the same, since we do not change existing assets. This way, we could actually make checking well-formedness more efficient, by checking only the new products, since the existing ones are exactly the same. Other templates that also deal with changing existing assets would have to consider the feature interaction problem when specifying its preconditions.

In practice, we need to rely on more flexible variations of ADD NEW OPTIONAL FEATURE that allow the association of more than one asset to the new optional feature. For brevity and simplicity, we omit the variations that would be useful to justify scenarios where many assets are introduced together with the new feature. Other templates specify the extension of a product line with other kinds of features (alternative, OR, etc.) (Neves et al., 2011). For example, ADD NEW MANDATORY FEATURE establishes that it is safe to extend the FM with a new mandatory feature, as long as we do not associate assets to it. The addition of such a feature that initially has no behavior or semantics might be an useful intermediate step during an evolution process, as we assess later.

3.2.2. Templates for annotative product lines

Although the templates introduced so far can be used for annotative product lines, they do not consider annotation mechanisms, such as preprocessing directives. To discover templates that deal with the specifics of such mechanisms, we analyzed the evolution history of the RGMS product line. A number of RGMS variations are implemented using conditional compilation as provided by the Velocity framework.[3] As a result of the RGMS analysis, we discovered a set of five templates and their variations, a subset of which we describe in this section. The description of the complete set is available in our online appendix.

Add new preprocessed feature. To specify that a new non mandatory feature can be safely implemented by preprocessing mechanisms, we introduce the ADD NEW PREPROCESSED FEATURE template in Fig. 7. It plays a similar role to ADD NEW OPTIONAL FEATURE. However, besides focusing on an annotative context, ADD NEW PREPROCESSED FEATURE applies also to alternative and OR features.

The template establishes that we can introduce a new feature *F* and annotated code *c* to an existing asset *a*, provided that the CK associates the added code with the new feature. We express this by associating a feature expression e', enabled by *F*, to the transformation that makes the preprocessing tag *x* true. Since we use the *x* tag to guard the added code *c*, this code is only enabled when e', and therefore *F*, is true. The *a*[ ] notation represents an asset with a specific place where code might be added, specifying a context surrounding the

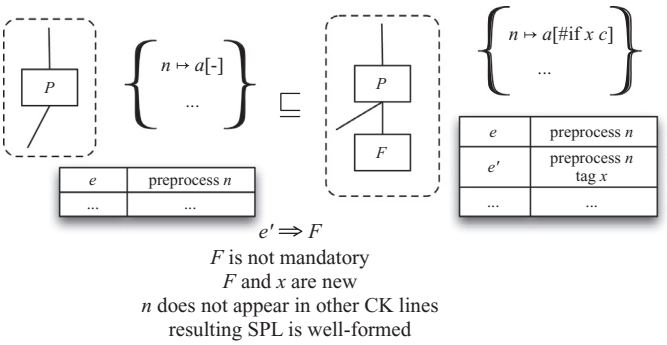


Fig. 7. ADD NEW PREPROCESSED FEATURE.

code to be added. The hyphen in *a*[−] indicates that no code is added, whereas *a*[#if *x c*] indicates that code *c* annotated with tag *x* appears in the place previously denoted by the hyphen. In the CK, we also see the preprocess transformation to *n*. This ensures that we preprocess the asset even when *e* is not true. These changes help to assure product line refinement, by making sure that the new code is only present in products that have the *F* feature, and that products built without the new feature correspond exactly to the original products.

Regarding the new feature *F*, the template only requires it to be new and non-mandatory. So, we can use the same template to add optional, alternative, and OR features. We could have done that too for templates in Section 3.2.1, to avoid having similar templates for each kind of feature. For simplicity, we opted to show first the specific template, which is easier to understand, and then present this more general format. This template also requires the initial CK to have a preprocess transformation associated with *n*, and therefore *a*[−]. This transformation often will not be in place if the asset does not contain preprocessing directives. But, in this case, we can use other templates to add the preprocessing transformation before applying ADD NEW PREPROCESSED FEATURE. This way we avoid creating a specific template variation for the case when the transformation is not in place.

Besides the preconditions discussed so far, ADD NEW PREPROCESSED FEATURE requires the *x* tag to be new. If *x* was already associated to another feature, for example, the new code *c* could be enabled even if *F* was not selected. We also need to guarantee that the resulting product line is well-formed. In fact, as we make no direct demands over *c*, we cannot know if the new products are well-formed. For most languages, this amounts to making sure that *a*[*c*] is a well-formed asset, but we prefer the more general condition.

To better explain this template, we illustrate its use in a concrete evolution scenario of the RGMS product line. Fig. 8 describes changes applied to a code asset as part of the effort to implement a new feature related to the research line followed by a member of a research group. To implement the feature, developers used a preprocessing directive to introduce feature related methods and fields to the class *Member*. As indicated by the internal box in the figure, these declarations instantiate the *c* meta-variable in the template, whereas `researchLine` instantiates *x*. In the template, we abstract the concrete syntax details of the particular preprocessing technology used by RGMS.

Looking to the product line as a whole, Fig. 9 describes also the changes made to the CK and FM. We see that developers added a new optional feature *ResearchLine*, under *RGMS*. Besides that, the new CK row indicates that the added field and method declarations should only be present when the *ResearchLine* feature is selected. Notice also that the first CK row now has a preprocess transformation to the *Member* asset, indicating that we applied PREPROCESS ASSET WITHOUT PREPROCESSOR DIRECTIVE before applying ADD NEW PREPROCESSED FEATURE. We could use ADD NEW PREPROCESSED FEATURE in this scenario because the added code does not cause compilation errors to the

[3] http://velocity.apache.org.

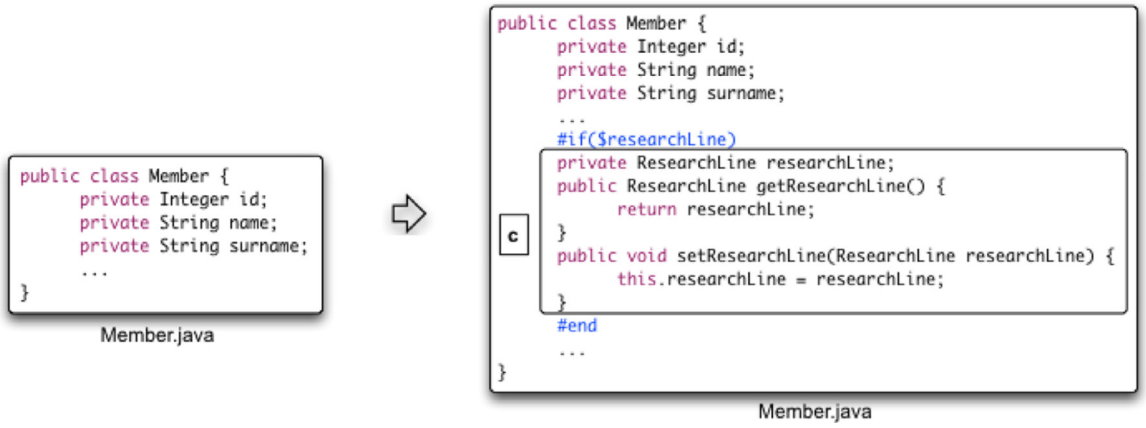


Fig. 8. ADD NEW PREPROCESSED FEATURE example — code assets.

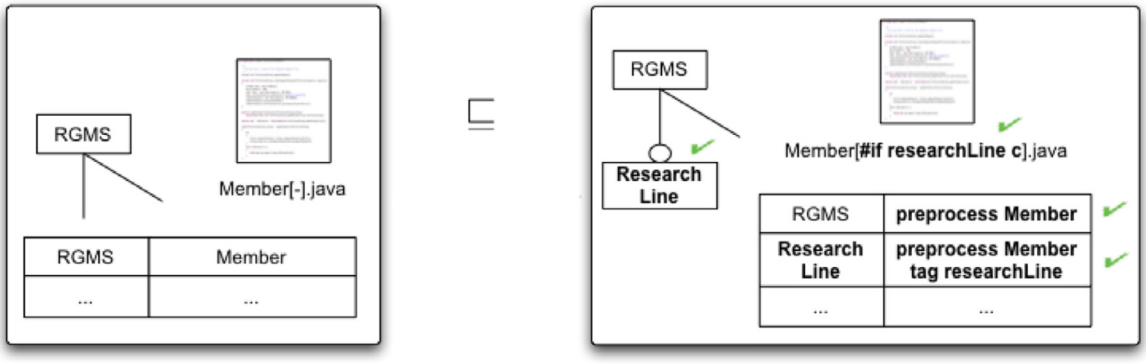
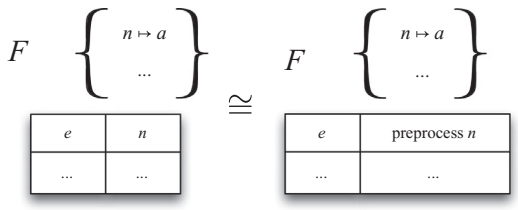


Fig. 9. ADD NEW PREPROCESSED FEATURE example — CK and FM.



(↔) *a* does not contain *#if's*

Fig. 10. PREPROCESS ASSET WITHOUT PREPROCESSOR DIRECTIVE.

Member asset; as a consequence, the resulting product line is well-formed. The other preconditions also hold in the analyzed scenario.

Preprocess asset without preprocessor directive. As an intermediate step when refining annotative product lines, it is useful to harmlessly introduce the *preprocess* transformation to CK items, as we discuss in the previous section. Fig. 10 precisely captures this situation. The template establishes that we can demand preprocessing of an asset *a* as long as this asset does not contain any preprocessing directives. In fact, preprocessing has no effect for this kind of asset. The evolution is safe because we preserve the behavior of *a* and, consequently, of all products that include *a*. Moreover, as mentioned in Section 2, the order we declare the transformations in the CK does not influence the generated products because, when evaluating the CK, we first consider all the *select and move components* transformations, then we set the tags values declared in the CK and finally we preprocess the required assets.

This template is useful to change a CK so that it matches other templates, such as ADD NEW PREPROCESSED FEATURE, that requires asset names associated to the *preprocess* transformation. So, it is simply an auxiliary template, but we show it here to illustrate another difference from the templates we presented so far. Note that we use ≅, instead of ⊑, to state that product line refinement holds in both directions. By applying the template from left to right, we introduce the *preprocess* transformation. In the opposite direction, we remove the *preprocess* transformation, given that it is not strictly necessary. As the template is bidirectional, we write '(↔)' before the precondition to denote that it is necessary when applying the template in both directions. In other templates, we use '(→)' before a precondition when it is only required from left to right. Similarly, we use '(←)' for conditions required only from right to left. This also helps to interpret each template as two product line refinement transformations with different preconditions. Some of the compositional templates we present in the previous section can be applied in both directions too, as we illustrate in our online appendix.

Other Annotative Templates. EXTRACT PREPROCESSED CODE is the counterpart of SPLIT ASSET for extracting preprocessed code from one asset and moving it to another asset. ADD DEAD PREPROCESSED CODE introduces a code fragment annotated with preprocessing directives to an asset. However, the segment is never present in preprocessed versions of the asset because we do not declare the tag associated to it in the CK. It abstracts some details and conditions from ADD NEW PREPROCESSED FEATURE. ADD NEW PREPROCESSED FEATURE requires that the new code fragment has to be associated in the CK to a new feature also introduced with the template application. It is a bidirectional transformation, so it defines that we can both add or remove code annotated with preprocessing directives from an asset as long as we respect the restrictions.

With ADD HARMLESS PREPROCESSING DIRECTIVE, it is possible to annotate an existing code fragment with a preprocessing directive and still preserve the asset behavior. To assure that the fragment is present

Table 1
Overview of product lines.

Product line	Type	Size	# Features	# Releases	# Scenarios
TaRGeT Main	Compositional	32 KLOC	42	4	9
TaRGeT Alt	Annotative	32 KLOC	46	2	4
MobileMedia	Compositional	3 KLOC	12	5	8
RGMS	Annotative	3 KLOC	20	13	13
Hephaestus-PL	Compositional	7 KLOC	11	3	5
SIGAA	Compositional	102 KLOC	16	6	15

in every version of the asset, we need to associate both the *preprocess* transformation and the tag declaration in the CK to the same feature expression that the asset was associated before. It is also bidirectional, given that we observe the restrictions. Finally, EXTRACT PREPROCESSED CODE is useful for situations where we want to change a feature variation implementation, for example when we want to extract the feature code annotated with preprocessing directives to an aspect (Alves et al., 2007) or to a super class. The template defines that it is possible to extract a code fragment annotated with preprocessing directives from an asset and place it in another new asset, as long as the set of modified assets refines the original asset.

4. Evaluation

To evaluate the expressiveness of our templates, we have conducted an empirical study that involved the analysis of the evolution history of five product lines. In this section, we detail the research questions and metrics adopted (Section 4.1), the assessment procedures and the target product lines (Section 4.2), the results and discussion of the study (Section 4.3), and its threats to validity (Section 4.4).

4.1. Research questions and metrics

To assess the expressiveness of the set of safe evolution templates described in the previous section, we analyzed part of the evolution history of five product lines of different domains. We have structured the study using the goal, question, metric (*GQM*) approach (Basili et al., 1994) to collect and analyze metrics related to the proposed templates.

Study main goal: Assess whether our templates are expressive enough and have the potential to provide guidance for product line developers to safely evolve a product line in intended safe evolution scenarios. Additionally, investigate whether they are also useful in unintended unsafe evolution scenarios.

Research question 1 (RQ1): Are the templates expressive enough to describe safe evolution scenarios from existing product lines?

Metric: Percentage of safe evolution scenarios that could be described by the existing templates. Number and percentage of templates applied to each different product line.

Research question 2 (RQ2): Could the templates have been used to detect unsafe evolution scenarios from existing product lines?

Metric: Number and percentage of unsafe evolution scenarios where an existing template could reveal an unsafe situation such as an invalid precondition.

4.2. Assessment procedures and overview of analyzed product lines

For each product line, we followed a process similar to the one used to discover templates, described in Section 3.1. First, we collected the version history of the product line and checked whether pairs of consecutive releases constituted or included safe evolution scenarios according to the product line refinement notion (Borba et al., 2012). For the pairs including safe evolution scenarios, we tried to find a template, or a list of templates that, when applied, could transform one release into its safely evolved counterpart.

For the unsafe evolution pairs, we investigated whether contextual information such as commit messages indicated that the evolution was supposed to be safe. When that was the case, we tried to find templates that could avoid the problem by revealing an invalid precondition, or structural constraints not matched by the concrete product line artifacts, in the specific scenario.

In our assessment, we studied the following product lines: TaRGeT, MobileMedia, RGMS, Hephaestus-PL (a product line of product line derivation tools), and the Library Module of SIGAA (an academic information system). As we have used two of them, TaRGeT and RGMS, for discovering templates (see Section 3.1), we chose non overlapping parts of their evolution history for the distinct activities of template discovery and assessment. Table 1 highlights the main characteristics of the evaluated product lines, such as size in KLOC, number of features, releases, and the number of safe evolution scenarios considered for the evaluation.

We decided to study these product lines for a number of reasons. First, they have been developed by disjoint developer teams but are nevertheless familiar to us: the first two authors were involved with the development of TaRGeT; most authors have used MobileMedia in other studies; RGMS was developed in a course offered by the second author; the third and fourth authors were lead developers of Hephaestus-PL; and the last two authors collaborate with the SIGAA team. This grants access to the source code repositories, and simplifies the analysis of the evolution histories, which often requires understanding the semantic effects of specific source code changes.

Second, TaRGeT, RGMS, Hephaestus-PL, and SIGAA repositories contain versions of their CK and FM, in similar formats. This helps because we are interested in analyzing the coevolution of assets, CK, and FM; it is harder to capture that from repositories containing only source code. MobileMedia's repository does not contain FM and CK, but, given its small size, we could infer different versions of these artifacts comparing the source code and extra information associated to each release.

Finally, the different domains, sizes, development teams, and variability implementation mechanisms used in the five product lines help to enrich the results of our study. TaRGeT, as described before, is a medium-sized product line of testing tools with branches developed by teams in different organizations. MobileMedia, although small and developed for academic purposes, focuses on the mobile application domain and uses implementation mechanisms rarely used by TaRGeT, such as aspect-oriented programming. It has also been used in a number of empirical studies about product lines, which helps to relate to our work. RGMS, contrasting with the previous two, is a web application and uses annotative variability mechanisms, which helps us to evaluate different kinds of templates. Additionally, since we actually have access to five RGMS alternative implementations by different student teams, we could observe the impact of development style on our results. SIGAA is a large web information system implemented in Java and used by a number of Brazilian universities. Finally, moving away from the Java domain, Hephaestus-PL is implemented in Haskell using rather different variability mechanisms.

Table 2
Number and percentage of templates applied to each different product line (Metric for answering RQ1). **T Main** is the main branch of TaRGeT; **T Alt** is the separate branch; **MM** is MobileMedia; **Hep** is Hephaestus-PL.

Template	T Main	T Alt	MM	RGMS	Hep	SIGAA	Total
Split asset	4 (13.7%)	4 (40%)	6 (25%)	1 (3.1%)	0 (0%)	13 (11.4%)	28
Refine asset	5 (17.2%)	0 (0%)	6 (25%)	0 (0%)	0 (0%)	71 (62.3%)	82
Refine asset variation	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	15 (13.2%)	15
Add new optional feature	5 (17.2%)	4 (40%)	4 (16.6%)	3 (9.3%)	1 (12%)	3 (2.6%)	20
Add new mandatory feature	3 (10.3%)	0 (0%)	1 (4.1%)	3 (9.3%)	0 (0%)	3 (2.6%)	10
Replace feature expression	3 (10.3%)	0 (0%)	4 (16.6%)	3 (9.3%)	0 (0%)	8 (7%)	18
Add new alternative feature	8 (27.5%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	8
Add new OR feature	1 (3.4%)	0 (0%)	2 (8.3%)	0 (0%)	7 (88%)	1 (0.9%)	11
Delete asset	0 (0%)	0 (0%)	1 (4.1%)	1 (3.1%)	0 (0%)	0 (0%)	2
(→) Preprocess asset without preprocessor directive	–	0 (0%)	–	0 (0%)	–	–	0
(←) Preprocess asset without preprocessor directive	–	0 (0%)	–	3 (9.3%)	–	–	3
Add dead preprocessed code	–	0 (0%)	–	0 (0%)	–	–	0
(→) Add harmless preprocessing directive	–	0 (0%)	–	4 (12.5%)	–	–	4
(←) Add harmless preprocessing directive	–	0 (0%)	–	3 (9.3%)	–	–	3
(→) Add new preprocessed feature	–	2 (20%)	–	6 (18.7%)	–	–	8
Extract preprocessed code	–	0 (0%)	–	5 (15.6%)	–	–	5

4.3. Results, analysis and discussion

This section reports the results of our study to assess whether our templates are expressive enough and have the potential to provide guidance for developers to safely evolve a product line. Our first and main focus is on the analysis of the intended safe evolution scenarios, observing if we can use the templates to describe safe evolution scenarios from the evaluated product lines (RQ1). Later we discuss the results related to the unintended unsafe evolution scenarios, to answer if we could use the templates to detect unsafe evolution scenarios from the evaluated product lines (RQ2).

4.3.1. Intended safe evolution scenarios

We first examined whether our templates were expressive enough to justify the modifications carried on in safe evolution scenarios from existing product lines. So, for each pair of product line versions representing a safe evolution scenario, we tried to find a list of templates that could transform the source version into the target version. We basically had to analyze source and target FM, CK, and asset mapping, and recursively check which templates matched the structure of these artifacts, and had valid preconditions for the specific context. For each scenario, we then obtain a list of templates that are able to describe the modifications. To understand the results in more detail, Table 2 presents the usage frequency of each template in TaRGeT's, MobileMedia's, RGMS', SIGAA's, and Hephaestus-PL's safe evolution scenarios.

The sum of the numbers in Table 2 is greater than the number of evaluated scenarios precisely because we often had to use more than one template to justify a single scenario. For example, in one of the TaRGeT's evolution scenarios, developers would have to first apply Add new mandatory feature to introduce the new feature *Word* in the FM, and then they would have to apply Replace feature expression to correctly associate the corresponding asset to the new feature. As another example, in one of Hephaestus-PL's evolution scenarios, developers would first have to use Add new optional feature to introduce the *Output Format* feature, and then Add new OR feature to add the new feature *Use Case to XML* as a subfeature of the former.

Similarly, we observe this in RGMS's scenarios. Some of them can only be justified by a mix of templates, including ones specific to annotative product lines. For example, in one scenario, the developers implemented support to the *MySQL* database as an alternative to *Postgres*. For this, they changed the *hibernatecfg.xml* file that stores information about database connections. Fig. 11 shows the *hibernatecfg.xml* file before and after the implementation of the *MySQL* feature. The developers introduced preprocessing directives to implement the associated variability. As a consequence, to justify the whole scenario, they had to change also the FM and CK, as illustrated in Fig. 12, which lists the templates that justify these changes. The developers first used Add harmless preprocessing directive to introduce the preprocessing directive to the *hibernateConfig* asset. Then they applied Add new mandatory feature and Replace feature expression to introduce the *Postgres* feature to the FM. Finally, they used Add new preprocessed feature to add the new alternative feature *MySQL* and to introduce the preprocessed code in the *hibernateConfig* asset. The *Postgres* and *MySQL* features are alternative, thus only one of them can be selected at a time.

We also observed that some scenarios demanded more than one application of the same template. For example, in scenarios that required extracting parts of a class to an aspect, it would be necessary to apply Split asset a number of times until the code was moved to the aspect. In such cases, we only list the template once, even if it is used more than once in the same scenario. This is because we intend to evaluate whether our set of templates is sufficient to express the safe evolution scenarios, regardless of the number of times each template is used. In Table 2, the rows with '–' indicate that we used TaRGeT, MobileMedia, and Hephaestus-PL to only evaluate the templates that apply to compositional product lines, as mentioned and justified earlier.

We also observe that in TaRGeT, Add new alternative feature was the most widely used in the analyzed 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). Moreover, during the evolution period we analyzed, the product line was continuously increasing the number of products and features. We think this is the reason why we did not find any occurrence of Delete asset. For the independently developed TaRGeT branch summarized in the third column, we analyzed only four safe evolution scenarios, which corresponded to the implementation of optional features. To implement these features, the product line was first restructured accordingly to Split asset, and then the features were implemented along the lines of Add new optional feature. Two of these scenarios also implemented new feature variations using preprocessing directives. For this kind of operation, the developers could have used Add new preprocessed feature.

Among MobileMedia's safe evolution scenarios, Split asset and Refine asset were the most used templates to justify the evolution scenarios. This is because the developers were refactoring the product line to support the implementation of new features. Split asset, together with object-oriented refactoring code templates, was used

Fig. 11. *hibernatecfg.xml* asset.

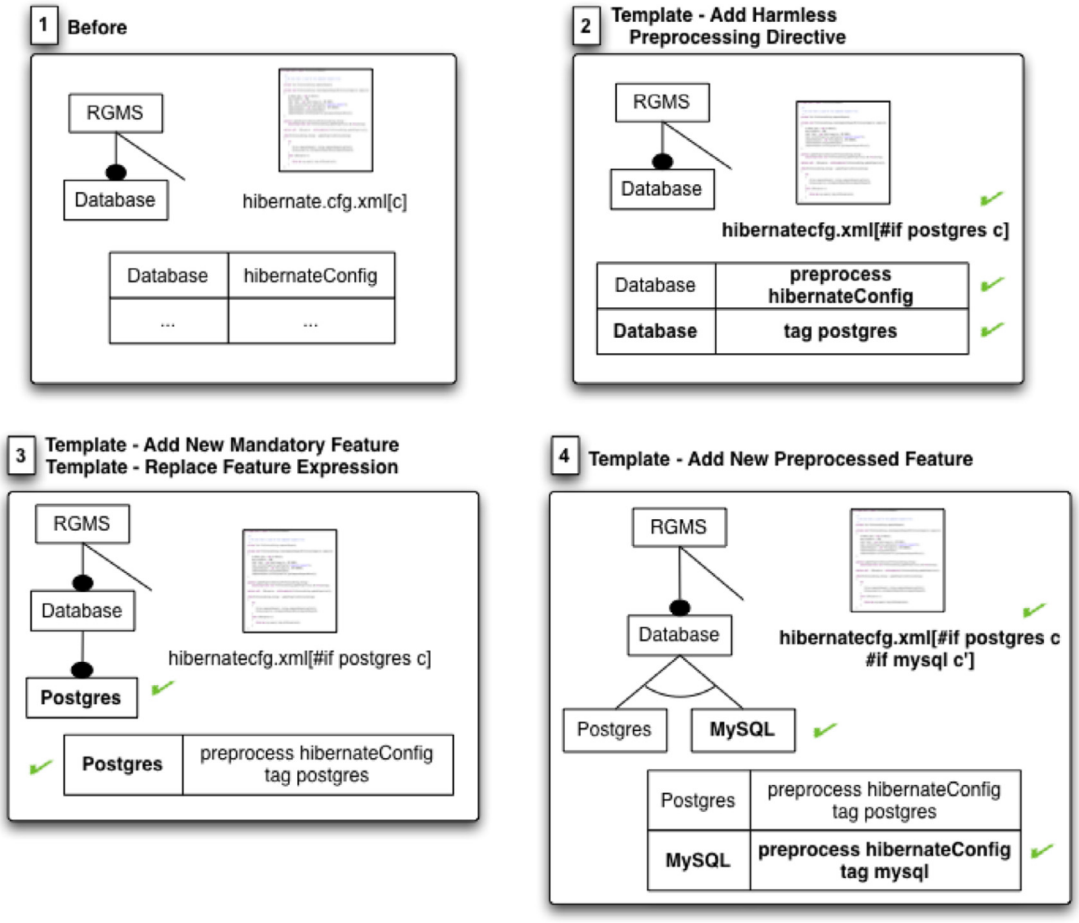


Fig. 12. *MySQL* feature evolution scenario.

to justify the extraction of subclasses, and also the extraction of class members to aspects, in both cases with the aim of implementing new optional features. REFINE ASSET was used to justify a scenario related to release 5, with the aim of structuring the product line for the introduction of a new OR feature, step later justified by ADD NEW OR FEATURE. Finally, DELETE ASSET was used once when an exception handling class became no longer useful because the exception cause no longer occurred.

When analyzing RGMS scenarios, we observed that ADD NEW PREPROCESSED FEATURE was the most frequently used. We believe that this was because in two RGMS releases, which amounts to 9 of the 13 scenarios used in the evaluation, we requested the students to

implement new features. Even knowing different variation implementation mechanisms, we suspect they preferred to use conditional compilation because it is less complex and does not require much effort to be implemented. We did not find any occurrence of Add new OR feature and Add new optional feature. Instead, we observed that Add new preprocessed feature was used to replace these templates, as it describes the same general modifications as the others, namely the introduction of a new non-mandatory feature to the product line. We did not found any occurrence of Refine asset because we found other transformation templates that could be used to deal with changes in assets, like Add dead preprocessed code, Add harmless preprocessing directive and Add new preprocessed feature. We derived Add dead preprocessed code abstracting some details and conditions from other transformation templates. We did not find any occurrences of it among RGMS scenarios but we decided to include this template in our set because we believe it is a representative operation that can be useful in other contexts different from the RGMS product line. Additionally, we did not find any occurrence of Preprocess asset without preprocessor directive but we decided to define it as a bidirectional refinement because it can be used as an intermediate template, to match the CK defined in Add dead preprocessed code, Add harmless preprocessing directive and Add new preprocessed feature.

In Hephaestus-PL, Add new OR feature was the most frequently employed, with 7 occurrences in all of the 5 safe evolution scenarios. This is due to the intended configurability of Hephaestus-PL's FM, in which the OR features represent assets and corresponding output formats, and the evolution scenarios consisted of increasingly providing support for these assets in such a way that any combination of them with corresponding output formats could be instantiated. Add new optional feature was applied once to add the *Output Format* optional feature before Add new OR feature could be employed as described previously. We have not identified any removal of features nor insertion of alternative features, nor any modifications that could be mapped to Split asset and Refine asset.

Lastly, Table 2 shows that six different templates were used in the evolution of the SIGAA Library module. Refine asset and its variations were the most used templates. They have been applied in two main kinds of scenarios: (i) when a new optional feature is added, the conditional execution expression is added in the code assets that can execute this feature. For example, the Terms of Agreement feature required the introduction of several conditional expressions into the JSF web pages and Java classes to determine its execution. This is the Refine asset template variation that we discuss in Section 3.2.1, which considers more than one asset refined at the same time. We need to use the variation, as simply adding new conditional statements, which are only activated when the new feature is selected, might not result in asset refinement, since the condition might evaluate to true and thus potentially change behavior. However, we also set the parameter for the new feature in the configuration file with the default value consisting of the feature not selected. Therefore, by changing both assets at the same time we do have refinement and existing products remain unchanged; (ii) the refactoring of existing features to accommodate new mandatory and optional features also required the modification of many code assets using Refine asset, as we show in Fig. 5, combined with Split asset. This strategy was used during the introduction of the Communication Note and System Management mandatory features, and also to accommodate the modifications related to user punishment (mandatory) and suspension (OR-feature), during the introduction of the Library Fine OR feature. This is the standard Refine asset template. The addition of new mandatory (Communication Note, User Punishment and System Management), optional (Book Reservation, Library Policy and Terms of Agreement) and OR (Library Fine) features required as expected the usage of the Add new mandatory feature, Add new optional feature, and Add new OR feature, respectively. In addition, the introduction of these new features also demanded the application of Replace feature expression to update the configuration knowledge for mapping the new features to their associated assets.

The lower presence of annotative templates in the analyzed product lines is given by two reasons. First, most of the evaluated product lines are compositional. Therefore, the annotative templates are not applicable in such cases. Second, the compositional templates are also useful in annotative product lines, since some of them do not detail particular changes to assets, such as Replace feature expression. Thus, they can be applied in both compositional and annotative product lines. Since RGMS is the only product line which is mostly implemented with annotative variability mechanisms, we observe differences in terms of template usage when comparing to the other product lines. TaRGeT's alternative branch has a few features implemented with annotations, but the vast majority of features is implemented with compositional mechanisms, as discussed. If we evaluate other annotative product lines, such as the Linux kernel or BusyBox, we would probably find more occurrences of annotative templates.

In summary, for all evaluated scenarios, in the five product lines, we found that our safe evolution templates, in addition to the FM refactorings listed in previous work (Alves et al., 2006), were sufficient to address the performed modifications. This reinforces the expressiveness of our templates, although further studies should certainly be carried on with other product lines and variation mechanisms.

Although we have not formally proved this, we believe that the evaluated scenarios in our studies could not be justified by alternative templates from our set. Two facts support this likely lack of bias in the number of occurrences of a given template. First, each template focuses on a different kind of change; this helps to avoid redundancy, and therefore alternative justification for the scenarios. Second, the templates were obtained by factoring out coarse-grained change scenarios, making each template concentrate on capturing fine-grained changes; so the template occurrences correspond to changes in small scale, which again offers less opportunities for using one template instead of another. Nevertheless, without further studies and formalization, we cannot claim that the template set is minimal or complete, since the templates have only been evaluated in five product lines and in specific scenarios. In particular, the fact that one template was not used does not imply that it is superfluous, since it could be useful in other scenarios of different product lines. Furthermore, we have observed, in some cases, recurring sequence of application of templates. However, we would need further studies to observe if such sequences happen systematically across different product lines.

4.3.2. Unintended unsafe evolution scenarios

Besides the safe evolution scenarios just discussed, during our study we also identified unintended unsafe evolution scenarios in the TaRGeT and MobileMedia product lines. These scenarios were supposed to be safe, but actually introduced defects, changing the behavior of existing products and, consequently, impacting users. Remembering our second research question (RQ2), we want to investigate whether our templates could be used to reveal invalid preconditions or structural constraints not matched by the concrete artifacts in such scenarios. In TaRGet, we know that these scenarios were unintendedly unsafe because we observed that these issues were later corrected in its evolution history. Moreover, commit messages mention these problems, indicating a different intention. The problems identified in the MobileMedia evolution are related to the safe composition property, discussed in detail in another work (Teixeira et al., 2013).

From the 20 safe evolution scenarios that we analyzed from TaRGeT main branch, two of them actually introduced defects to the product line. These scenarios occurred in the template discovery step of TaRGeT's evolution. We previously described one of these problems (Neves et al., 2011), namely where an optional feature was added but incorrectly referred in the CK. The other defect was similar to this

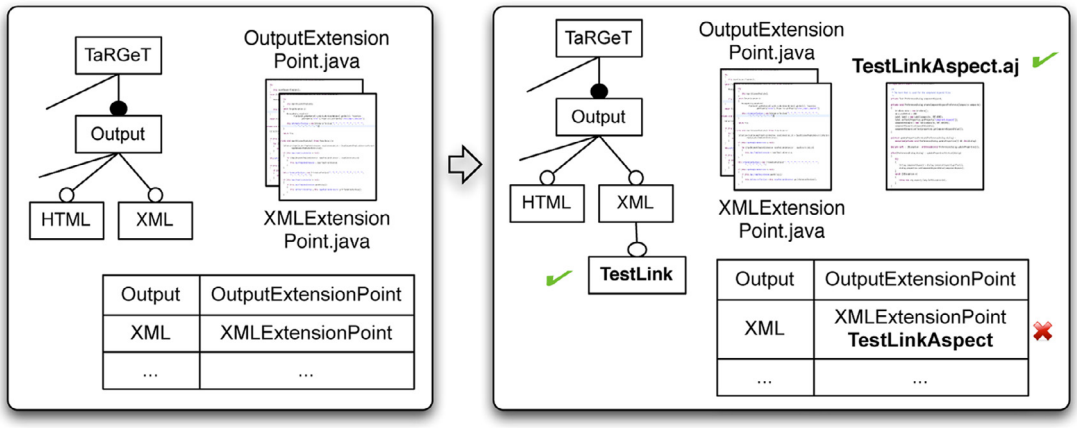


Fig. 13. *TestLink* feature evolution scenario.

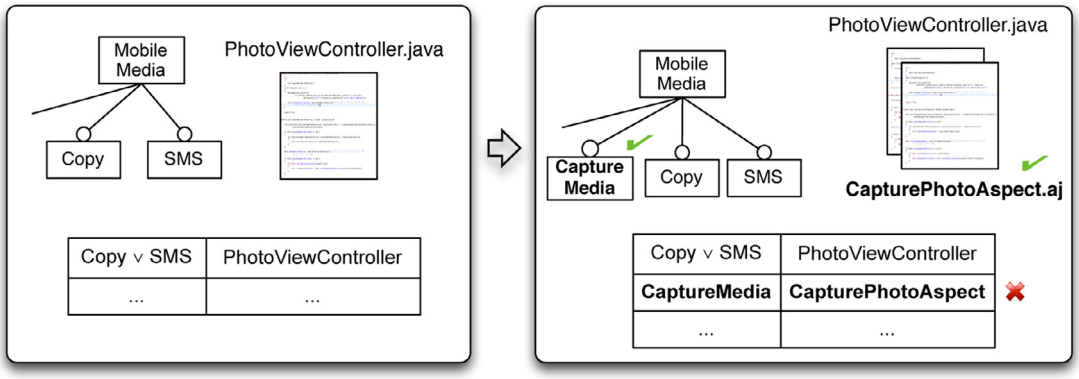


Fig. 14. *CaptureMedia* feature evolution scenario.

one and is illustrated in Fig. 13. In this example, the *TestLinkAspect* is responsible for implementing a feature that enables the generation of test suites in a specific *XML* format. This aspect was incorrectly associated to the *XML* feature expression in the CK, making all products having the *XML* feature to generate only test suites in the *TestLink* format. This problem could have been avoided using the ADD NEW OPTIONAL FEATURE template.

In MobileMedia we found four cases that change the behavior of existing products when that was not the intention (Teixeira et al., 2013). Fig. 14 describes one of these cases, where some fields of the *PhotoViewController* class were extracted to the *CapturePhotoAspect* aspect. However, this aspect was incorrectly associated to the *CaptureMedia* feature in the CK, while the original *PhotoViewController* class is associated to the feature expression *Copy* ∨ *SMS*. This problem could be avoided by using SPLIT ASSET. The other defects are about incorrect association of feature expressions in the CK, which could be avoided using REPLACE FEATURE EXPRESSION. We did not find any occurrence of this kind of problem in the RGMS scenarios. We think that, as we only analyzed the code from the main releases, the students may have fixed these problems before the release. Similarly, we have not observed such issues in Hephaestus-PL, nor in SIGAA. Table 3 summarizes the answer to RQ2, by showing the number and percentage of unsafe evolution scenarios where an existing template could reveal an invalid precondition for each product line.

4.3.3. Discussion

Although the evaluation so far considers only Java, AspectJ, and Haskell code assets, our templates and the product line refinement notion that supports them are not limited to a specific kind of asset.

Table 3
Number and percentage of unintended unsafe evolution scenarios in each product line (Metric for answering RQ2).

Product line	Total of scenarios	Unsafe evolution scenarios found
TaRGeT main	20	2 (10%)
MobileMedia	8	4 (50%)
RGMS	15	0 (0%)
Hephaestus-PL	5	0 (0%)
SIGAA	15	0 (0%)

That is why we can use the templates in other contexts than the ones we illustrate here. Nonetheless, the product line refinement notion relies on a notion of asset (behavioral) refinement. In fact, some of our templates have restrictions associated to asset refinement (for example, see SPLIT ASSET). So, for each kind of asset to be considered, one should choose a proper asset refinement notion. Such a notion can be more or less restrictive, ranging from formal programming transformation laws (Borba et al., 2004) to tests as in Safe Refactor (Soares et al., 2010). The only requirement is that these notions must be preorders and satisfy a compositionality condition (Borba et al., 2012).

Besides that, there should also be a notion of asset well-formedness, as discussed in Section 2.2. The templates assume well-formedness of the source product line, and in some cases, when extending the product line, they require, as a precondition, well-formedness of the resulting product line. This relates to safe composition (Teixeira et al., 2013; Thaker et al., 2007) and associated approaches that efficiently check that. In templates that add features, we

only need to check that the new products are well-formed, since the existing products, which are already well-formed, remain the same. Moreover, for all other templates that do not require this precondition, we prove that the resulting product line is well-formed (Borba et al., 2012).

Contrasting with the asset language independence, our templates are specific to the FM and CK languages we use here. This should be expected since the templates refer to the specific syntax and rely on semantic details of these languages. Although some of the templates could be easily adapted to alternative languages and notations, including decision models (Czarnecki et al., 2012; Weiss et al., 2008), we do not explore this possibility since our focus is to understand and explain the evolution history of product lines based on the languages we consider here. The product line refinement notion, on the other hand, is independent of such languages and could be used to justify safe evolution templates specific to the alternative notations we just mentioned. Nonetheless, we acknowledge that formally checking refinement is hard. When deriving the templates from concrete evolution scenarios, we first informally checked that they complied with the refinement notion that we rely on. Later some of us formally proved soundness of the templates for product lines (Borba et al., 2012; Teixeira, 2014), but we consider this out of the scope of this work.

In the concrete evolution scenarios we analyzed, it was often necessary to combine more than one template to justify the safety of the scenario. For example, Add new mandatory feature and Replace feature expression, and Preprocess asset without preprocessor directive and Add dead preprocessed code (using the transformation from right to left), were often used in sequence, since the preconditions were satisfied. The transitivity property of the product line refinement notion (Borba et al., 2012) allows the application of different templates in sequence, resulting in a well-formed, refined, product line.

In the online appendix, we specify all templates in detail, detailing the conditions for applying them in both directions, when applicable. This way, one can precisely implement tool support for applying them to concrete product lines. The details are also important to guide developers to safely evolve product lines. For conciseness and abstraction, we use a declarative approach to describe the templates. For example, we could use the FLiP tool (Alves et al., 2008) to implement support for applying templates in a semi-automatic way. This tool already supports product line extraction and evolution tasks, focusing mostly on code. Therefore, we would need to extend it so it could also consider refinement templates such as the ones presented here, that also change FM and CK. A particular issue is verifying the preconditions for applying a template. In the particular cases where we require well-formedness of the resulting product line, such as Add new optional feature, for example, we can rely on safe composition approaches, as discussed. When a precondition requires asset refinement, we can approximate this using product line refinement checkers (Ferreira et al., 2014), to improve confidence that the template implementation does not introduce behavioral changes.

Finally, after observing the applicability of the templates to a myriad of assets as previously mentioned, we found no relationship between the template type (compositional versus annotative) and its applicable abstraction level (architectural versus detailed design). Nonetheless, even without choosing a particular asset language, annotative templates can already capture finer-grained asset changes. So, for instance, considering that an asset is defined at architectural design, annotative templates can also address issues therein, provided that these have an annotative nature. On the other hand, if code artifacts are considered, annotative templates can be used to capture finer-grained changes, but compositional templates are also useful, especially for feature addition (for example, Add new optional feature).

4.4. Threats to validity

In this section we discuss the threats to the validity of our study.

Conclusion validity. Our approach is motivated by the assumption that evolving product lines without guidance is more error-prone. Although our main focus here is not to evaluate that, we found preliminary evidence of this issue when identifying evolution scenarios that were supposed to preserve the behavior of existing products but did not preserve. However, we found these issues by simply inspecting the product line artifacts. Despite our familiarity with the product lines, more precise results could be obtained by collecting evidence by testing products. We might have overlooked such issues, thus the number of defects could be higher, which would more emphatically answer RQ2. The contextual information we relied on to judge the intention of the evolution is another potential threat, but it was clear enough in the cases we observed. So, again, mitigating these threats could only reinforce the obtained results.

Another threat coming from the manual analysis during the identification and evaluation of the templates is that we could have made mistakes when verifying the preconditions necessary to apply a template correctly. This could impact our study by changing the templates frequency results we presented, or even providing evidence of scenarios that could not be handled by our templates. These situations were, however, often discussed by more than one author.

Internal validity. When analyzing RGMS evolution, we observed that, in some occasions, different pairs of students used the same template combination in similar scenarios. For example, to support a new database management system, four pairs of students implemented it by changing the Hibernate configuration file, which stores the configuration parameters of each database, and used conditional compilation to define which database would be available in a product. This may have happened because the students discussed with each other how to implement the feature and then they all decided to implement it using the same technique. While this can represent a threat to the internal validity of our study, at most it restricted the analysis of other different template combinations when implementing variability strategies.

It is important to mention that using the same product line to both infer and evaluate the templates could be a threat, but we divided the scenarios we identified in two different groups. We used one of these groups to infer the templates and another group, with different scenarios, to evaluate them. In the TaRGeT product line, we used the scenarios implemented from release 4.0 to 5.0 to infer the templates and the ones implemented from release 5.0 to release 6.0 in our evaluation. We selected two RGMS scenarios to infer the templates and 13 other scenarios to evaluate the templates. MobileMedia's, SIGAA's and Hephaestus-PL's scenarios were only used to evaluate the templates. Moreover, as illustrated later, we believe that the observed product line templates depend more on the demanded tasks (feature creation, modification, etc.) than on the people that performed these tasks. This further reduces the possibility of threat for using the same product line to infer and evaluate templates.

External validity. Another threat to our work is the fact that MobileMedia and RGMS are small product lines developed for research and educational purposes, respectively, so the scenarios identified in their evolution could be simple and require the use of few templates. However, we observed that these scenarios usually required the combination of different templates. In addition, most of the identified templates were also found in the analysis of other product lines, thus reinforcing their relevance and applicability.

When choosing the studied product lines, we did not aim for representativeness. We tried to obtain some diversity, but not in a systematic way as described in (Nagappan et al., 2013). Because of the limited quantity and nature of the analyzed product lines, and the chosen releases and scenarios, our quantitative results cannot be generalized to other product lines, specially the annotative templates. To

further evaluate those, we would need to consider additional annotative product lines. However, the qualitative results bring evidence that our set of safe evolution templates is expressive at least in a limited context. We believe that factors such as product line domain, and team experience and motivation, impact less on our expressiveness results, but we nevertheless studied product lines of different domains, developed by disjoint developer teams. The kinds of templates used seem to depend more on factors such as variability implementation mechanism and performed tasks. For example, as the early history of a product line might involve a lot of feature creation tasks, analyzing only early histories might reveal only templates such as ADD NEW OPTIONAL FEATURE. Although we tried to mitigate this kind of threat, further studies would be useful to complement our results. Nevertheless, by studying aspects of the evolution of a few systems, we hope to be contributing to the body of evidence on product line evolution.

The product line refinement theory does not aim to cover all kinds of evolution tasks, leaving out certain kinds of useful changes. Nonetheless, studies about the evolution of the Linux kernel FM (Dintzner et al., 2013; Lotufo et al., 2010) show that most of the changes consist of adding new features and modifying existing features, while a smaller number of those changes correspond to feature removals. Therefore, the aim of this work is to provide support for safely evolving a product line, to avoid problems that can be introduced when performing such tasks.

5. Related work

The work reported here is an extension of our previous investigation on how compositional software product lines safely evolve (Neves et al., 2011). Here we go beyond that by analyzing annotative product lines (Kästner et al., 2008), in particular the ones that manage variability with preprocessing mechanisms. As a result, we discover and catalog new transformation templates that deal with a more expressive CK notion that maps feature expressions to asset transformations (including preprocessing). This way we provide support for evolving both compositional and annotative product lines, and even hybrids. Besides that, here we analyze the evolution history of two additional product lines, including one that is largely annotative. Based on that, we bring additional evidence of the expressiveness of the proposed templates, including the new ones for annotative product lines.

Passos et al. (2013) analyze the evolution of the Linux kernel, proposing evolution patterns similar to our templates, taking into account changes to the FM (Kconfig, in their context), CK (Kbuild), and assets with preprocessing directives (cpp). These changes are not restricted to safe evolution scenarios, so they also consider potentially unsafe transformations. Nonetheless, we see the studies as complementary, as some of their reported patterns are similar to our templates, such as ADD NEW OPTIONAL FEATURE. The main difference is that our intention is to provide guarantees and support when the intention is to safely evolve a product line, whereas they are interested to report the evolution scenarios that happen in a particular timeframe.

Seidl et al. (2012) present an evolution system for model-based product lines. As in our work, they represent product lines using three spaces, namely problem (FM), mapping (CK), and solution (code assets) space. They aim to keep consistency of the spaces when performing an evolution operation that can potentially harm the mapping between problem and solution space. To achieve that, they present and evaluate a number of semi-automatic and syntactic remapping operators that support developers on this task. Although they express no explicit concern with safe evolution, nor propose overall product line transformation templates, one could likely use their operators to implement the CK changes expressed by the templates we present here.

Schulze et al. provide a definition for variant-preserving refactorings in the context of feature-oriented product lines (Schulze et al., 2012). Moreover, they also propose a refactoring catalog, illustrating four different refactorings based on extensions of object-oriented transformations (Fowler, 1999). Later, they provide tool support to provide an implementation of such refactorings, so they can be applied semi-automatically (Schulze et al., 2013a). These works are complementary to ours, in the sense that they could be seen as concrete instances of the product line refinement theory, for particular FM, CK, and asset languages different from those we focus on this work. The variant-preserving refactoring definition (Schulze et al., 2012) could also be seen as a particular refinement notion for the context of feature-oriented product lines. Future work could formally relate both works.

Schulze et al. also propose a refactoring catalog for delta-oriented (DOP) product lines (Schulze et al., 2013b), implemented in the context of an Eclipse plugin. They propose code smells that might be useful for identifying opportunities for applying product line refactorings. Some of the proposed refactorings are correspondent to templates we present here, such as REPLACE FEATURE EXPRESSION. Although they do not provide a formalization, their work is complementary to ours, by providing an implementation of such transformations in the context of a particular asset language.

Heider et al. (2012) propose a regression testing approach to detect the impact of changes to variability models. The basic idea is to identify existing products that are affected by evolving an associated product line. The illustrated implementation compares, in a lexical or syntactical way, the assets of the products generated by the original and the new (evolved) product line. If there are differences, the tool warns the user. This might be helpful to identify potentially unsafe modifications, but that is not explored by the paper. The tool just indicates what has to be analyzed, with no information on whether the modification is safe or not. Similarly to our work, they are interested in analyzing impact on the existing products. Differently from our work, they do not consider all possible products in the FM, but rather, only the product configurations previously created and derived. More important, they perform analysis *a posteriori*, that is, after a change happens, and only with the aim of *warning* developers. We propose templates to, possibly *a priori*, prevent developers of making changes that impact existing products. Based on the same underlying theory we rely on, but following an *a posteriori* approach, Ferreira et al. (2014) detect unsafe transformations by applying regression unit testing; this contrasts with the more efficient but less precise lexical or syntactical approach of Heider et al.

The notion of product line refinement discussed here first appeared with a refactoring focus (Borba, 2009), illustrating different kinds of refactoring transformation templates that can be useful for deriving and evolving product lines. Borba et al. (2012) extend and generalize the initial formalization, establishing interfaces between the product line refinement theory and the languages used to describe product line artifacts. They also instantiate the theory with the formalization of specific languages for typical product line artifacts (FM and CK), and introduce and prove soundness of a number of associated product line refinement transformation templates. Here we rely on the existing definition of product line refinement, and the idea of safe evolution transformation templates. However, we go further by proposing new templates for a different CK language, based on the empirical analysis of the evolution history of two real product lines, and evaluating both old and new templates in five product lines. Rubin and Chechik (2012) use the refinement theory for combining products to generate product lines. They present a *merge-in* operator that adds a product (actually a model in their context) to an existing product line. They formally prove the correctness of this operator using the product line refinement notion, showing that the resulting product line generates the same products as before, plus the additional merged model. We can view

their operator as a different way of expressing ADD NEW OPTIONAL FEATURE.

Alves et al. (2006) propose an informal refactoring notion for product lines, and propose a catalogue of FM refactorings. Gheyi et al. (2008) extend that by proposing a complete and minimal catalog of FM transformation templates that preserve the set of FM valid configurations. We go beyond that because the product line refinement notion that we rely on, and consequently our templates, support other product line artifacts like CK and assets, in addition to FM. However, as the product line refinement notion is compositional (Borba et al., 2012), we can use the FM templates proposed by Gheyi et al. together with our safe evolution templates to independently evolve FMs in a safe way. With a similar focus on FMs only, Lotufo et al. (2010) analyze 21 versions of the Linux kernel over five years. They analyze how a number of characteristics, such as number of features, height of the tree and depth of the leaves, evolve through versions. Based on this investigation, they identify challenges encountered in the process. However, as with the studies just mentioned, they only focus on the FM, and do not take into account the other artifacts.

Several approaches (Kaestner et al., 2007; Kolb et al., 2005; Liu et al., 2006; Trujillo et al., 2006) focus on refactoring a product into a product line, not exploring product line evolution in general, as we do here with our templates. First, Kolb et al. (2005) discuss a case study in refactoring legacy code components into a product line implementation. They define a systematic process for refactoring products with the aim of obtaining product line assets. There is no discussion about FMs and CK. Similarly, Kaestner et al. (2007) focus only on transforming code assets, implicitly relying on refinement notions for aspect-oriented programs. As discussed here and elsewhere (Borba, 2009), these are not adequate for justifying product line refinement. Trujillo et al. (2006) 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. (2006) 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 a notion of asset refinement, both theories could be used together.

Thüm et al. (2009) 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 go beyond FMs and also take into account edits in other artifacts like CK and code assets. Moreover, we are interested in providing guarantees, *a priori*, that particular kind of edits result in safe evolution of the product line. So, using their classification, we are more interested in refactorings and generalization edits, not considering, except for pathological cases, specialization and arbitrary edits.

Murphy-Hill et al. (2012) present a study about how programmers refactor. They analyze usage data from Eclipse users and refactoring tools logs to classify which programs refactorings are more frequently used. They also use commits from Eclipse Concurrent Versioning System (CVS) repositories and they infer which refactorings were performed by comparing adjacent commits manually. In this work we use a similar strategy to discover product line safe evolution scenarios and to infer which templates were used in these scenarios. We also classify which templates are more commonly used using data retrieved from the evolution of five product lines.

Weissgerber and Diehl (2006) propose a technique to detect changes that are likely to be refactorings. Based on information from code repositories, they reconstruct refactorings by means of syntactical and signature-based analysis, using code-clone detection to refine the results, ranking the refactoring candidates. Their technique focuses only on the code, while our analysis must take into account the whole product line, going beyond code assets. Therefore, this is why we conduct a manual analysis. Nonetheless, both approaches are complimentary, and could be integrated for cases where only the code has changed. As an example, we could use their technique to detect instances of the REFINE ASSET template.

Thaker et al. (2007) define that safe composition is related to safe generation and the verification of properties for product line assets. They showed how product line type safety properties can be verified using FMs and SAT solvers. Teixeira et al. (2013) present an approach to verify safe composition of compositional product lines. As some of our templates require the resulting product line to be well formed, we can use the safe composition approach to verify this type of restriction.

6. Conclusion

In this work we investigate the safe evolution of both compositional and annotative product lines (Kästner et al., 2008). By analyzing concrete safe evolution scenarios and the associated changes to the different product line artifacts (feature model, configuration knowledge, and assets), we were able to propose a set of safe evolution templates that can be used by developers in charge of maintaining product lines. Besides being in accordance with the product line refinement notion that we rely on (Borba et al., 2012), the described templates abstract, generalize, and factorize the analyzed scenarios. This way they give developers confidence that product line transformations are safe, in the sense that the behavior of existing products is preserved. If applied in retrospective, they can even help to find unintended unsafe modifications to a product line. The templates can also be used as a basis to automate support for safe product line evolution.

To evaluate the proposed templates, we analyzed part of the evolution history of five product lines. We could observe that our templates can address the safe modifications that developers performed in the analyzed scenarios. This corroborates the expressiveness of our template set, although it was not our aim to obtain a complete (in the relative sense of previous algebra of programming results (Borba et al., 2004)) or canonical set. More detailed investigations are needed to properly explore these properties. We could also observe that if the templates had been used as a guide when evolving the product lines, they could have helped to avoid the evolution errors that we identified during our analysis. These errors are modifications that were supposed to be safe, but actually changed the behavior of existing products. We believe that some of these problems could be avoided with a more rigorous regression testing process. However, it is often expensive to generate and test all product line products. Soundness of the templates is proved elsewhere (Borba et al., 2012).[4]

Our results also show evidence that product line manual evolution can be time consuming, because it usually involves the analysis and modification of many source code assets, in addition to the feature model and configuration knowledge. We believe that the automating the templates with a development tool could help to address this

[4] PVS specification and proof files available in http://twiki.cin.ufpe.br/twiki/bin/view/SPG/TheorySPLRefinement.

issue, but this should be further explored, given recent evidence about the difficulties for using refactoring tools (Murphy-Hill et al., 2012).

Although our templates focus on specific feature model and configuration knowledge notations, the underlying refinement theory (Borba et al., 2012) is not language specific. Decision models could be mapped to our feature model notation (Czarnecki et al., 2012). More elaborate feature model and decision model notations with cardinality and attributes would require extra templates; the existing ones would still hold. Existing notion of product line refinement applies for any feature modeling notation that describes (possibly infinite) sets of product configurations. We only need to instantiate it with a notation that expresses its semantics as a set of configurations. The same happens for more elaborate CK notation exploring cardinality and attribute information, such as Kconfig and Kbuild (Lotufo et al., 2010; Passos et al., 2013), which would require extra templates. In all of these approaches, there are ways to generate a specific product according to a valid configuration. As long as such a function exists, we can instantiate the refinement theory with the corresponding CK approach, and thus, derive more templates specific to that language.

We also plan on investigating how our theories can provide support even in the context of unsafe evolution scenarios, such as retirement. For example, evolution patterns from the Linux kernel (Passos et al., 2013) show that feature retirements happen. The patterns that remove features consist of removing a feature from all variability spaces. This way, we could establish an alternative refinement notion, requiring that all products modulo the removed feature should be refined by the resulting product line.

Acknowledgments

We thank colleagues of the Software Productivity Group,[5] especially Rohit Gheyi, Fernando Castor, Raphael Oliveira, André Lanna, and Idarlan Machado for important feedback and fruitful discussions about this work. For partial financial support, we would like to thank the National Institute of Science and Technology for Software Engineering (INES) and the Brazilian research agencies CNPq, CAPES and FACEPE.

References

Alves, V., Calheiros, F., Nepomuceno, V., Menezes, A., Soares, S., Borba, P., 2008. Flip: Managing software product line extraction and reaction with aspects. In: SPLC, p. 354.
Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., Lucena, C., 2006. Refactoring product lines. In: GPCE, pp. 201–210.
Alves, V., Matos, P., Cole, L., Vasconcelos, A., Borba, P., Ramalho, G., 2007. Extracting and evolving code in product lines with aspect-oriented programming. Trans. Aspect-Oriented Software Dev. 4, 117–142.
Basili, V.R., Caldiera, G., Rombach, H.D., 1994. The goal question metric approach. Encyclopedia of Software Engineering. Wiley.
Borba, P., 2009. An introduction to software product line refactoring. In: GTTSE Summer School.
Borba, P., Sampaio, A., Cavalcanti, A., Cornelio, M., 2004. Algebraic reasoning for object-oriented programming. Sci. Comput. Program. 52, 53–100.
Borba, P., Teixeira, L., Gheyi, R., 2012. A theory of software product line refinement. Theor. Comput. Sci. 455, 2–30.
Clements, P., Northrop, L., 2001. Software Product Lines: Practices and Patterns. Addison-Wesley.
Czarnecki, K., Eisenecker, U., 2000. Generative Programming: Methods, Tools, and Applications. Addison-Wesley.
Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K., Wasowski, A., 2012. Cool features and tough decisions: a comparison of variability modeling approaches. In: VaMoS, pp. 173–182.
Czarnecki, K., Pietroszek, K., 2006. Verifying feature-based model templates against well-formedness OCL constraints. In: GPCE, pp. 211–220.
Dintzner, N., Deursen, A.V., Pinzger, M., 2013. Extracting feature model changes from the linux kernel using fmdiff. In: VaMoS, pp. 22:1–22:8.
Ferreira, F., Gheyi, R., Borba, P., Soares, G., 2014. A toolset for checking SPL refinements. J. Univers. Comput. Sci. 20 (5), 587–614.
Fowler, M., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley.
Gacek, C., Anastasopoules, M., 2001. Implementing product line variabilities. In: SSR, pp. 109–117.
Gheyi, R., Massoni, T., Borba, P., 2008. Algebraic laws for feature models. J. Univers. Comput. Sci. 14 (21), 3573–3591.
Heider, W., Rabiser, R., Grünbacher, P., Lettner, D., 2012. Using regression testing to analyze the impact of changes to variability models on products. In: SPLC. ACM, pp. 196–205.
Kaestner, C., Apel, S., Batory, D., 2007. A case study implementing features using AspectJ. In: SPLC, pp. 223–232.
Kaestner, C., Apel, S., Rahman, S.S.u., Rosenmueller, M., Batory, D., Saake, G., 2009. On the impact of the optional feature problem: analysis and case studies. In: SPLC, pp. 181–190.
Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.S., 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report. CMU/SEI-90-TR-21, SEI, CMU.
Kästner, C., Apel, S., Kuhlemann, M., 2008. Granularity in software product lines. In: ICSE, pp. 311–320.
Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W., 2001. Getting started with AspectJ. Commun. ACM 44 (10), 59–65.
Kolb, R., Muthig, D., Patzke, T., Yamauchi, K., 2005. A case study in refactoring a legacy component for reuse in a product line. In: ICSM, pp. 369–378.
Krueger, C., 2002. Easing the transition to software mass customization. In: PFE, pp. 282–293.
van der Linden, F., Schmid, K., Rommes, E., 2007. Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering. Springer.
Liu, J., Batory, D., Lengauer, C., 2006. Feature oriented refactoring of legacy applications. In: ICSE, pp. 112–121.
Lotufo, R., She, S., Berger, T., Czarnecki, K., Wasowski, A., 2010. Evolution of the Linux kernel variability model. In: SPLC, pp. 136–150.
Murphy-Hill, E., Parnin, C., Black, A., 2012. How we refactor, and how we know it. IEEE Trans. Software Eng. 38, 5–18.
Nagappan, M., Zimmermann, T., Bird, C., 2013. Diversity in software engineering research. In: ESEC/FSE, pp. 466–476.
Neves, L., Teixeira, L., Sena, D., Alves, V., Kulezsa, U., Borba, P., 2011. Investigating the Safe Evolution of Software Product Lines. GPCE. ACM, pp. 33–42.
Passos, L., Guo, J., Teixeira, L., Czarnecki, K., Wasowski, A., Borba, P., 2013. Coevolution of variability models and related artifacts: a case study from the linux kernel. In: SPLC, pp. 91–100.
Pohl, K., Böckle, G., van der Linden, F., 2005. Software Product Line Engineering: Foundations, Principles and Techniques. Springer.
Roberts, D.B., 1999. Practical Analysis for Refactoring (Ph.D. thesis). University of Illinois.
Rubin, J., Chechik, M., 2012. Combining related products into product lines. In: FASE, pp. 285–300.
Schobbens, P.-Y., Heymans, P., Trigaux, J.-C., Bontemps, Y., 2007. Generic semantics of feature diagrams. Int. J. Comput. Telecommun. Networking 51 (2), 456–479.
Schulze, S., Lochau, M., Brunswig, S., 2013a. Implementing refactorings for FOP: lessons learned and challenges ahead. In: FOSD, pp. 33–40.
Schulze, S., Richers, O., Schaefer, I., 2013b. Refactoring delta-oriented software product lines. In: AOSD, pp. 73–84.
Schulze, S., Thüm, T., Kuhlemann, M., Saake, G., 2012. Variant-preserving refactoring in feature-oriented software product lines. In: VaMoS, pp. 73–81.
Seidl, C., Heidenreich, F., Assmann, U., 2012. Co-evolution of models and feature mapping in software product lines. In: SPLC, pp. 76–85.
Soares, G., Gheyi, R., Serey, D., Massoni, T., 2010. Making program refactoring safer. IEEE Software 27 (4), 52–57.
Teixeira, L., March 2014. Safe Evolution of Software Product Lines and Sets of Product Lines (Ph.D. thesis). Federal University of Pernambuco.
Teixeira, L., Borba, P., Gheyi, R., 2013. Safe composition of configuration knowledge-based software product lines. J. Syst. Software 86 (4), 1038–1053.
Thaker, S., Batory, D., Kitchin, D., Cook, W., 2007. Safe composition of product lines. In: GPCE, pp. 95–104.
Thüm, T., Batory, D., Kästner, C., 2009. Reasoning about edits to feature models. In: ICSE, pp. 254–264.
Trujillo, S., Batory, D., Diaz, O., 2006. Feature refactoring a multi-representation program into a product line. In: GPCE, pp. 191–200.
Weiss, D.M., Li, J.J., Slye, J.H., Dinh-Trong, T.T., Sun, H., 2008. Decision-model-based code generation for SPLE. In: SPLC, pp. 129–138.
Weissgerber, P., Diehl, S., 2006. Identifying refactorings from source-code changes. In: ASE, pp. 231–240.

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

Laís Neves is a software developer and holds a MSc degree in Computer Science from the Federal University of Pernambuco, Brazil. Her main research interests are software product lines and software development.

Paulo Borba is a Professor of Software Development at the Informatics Center of the Federal University of Pernambuco, Brazil, where he leads the Software Productivity Group. His main research interests are in the following topics and their integration: software modularity, software product lines, and refactoring.

Vander Alves is a Professor Adjunto 3 (tenured) at the Computer Science Department of University of Brasilia, Brazil. He conducts research on Software Product Lines, Ambient Assisted Living, and Command and Control. Previously he held research and development positions at Fraunhofer IESE, Lancaster University, and IBM Silicon Valley Lab.

Lucinéia Turnes is a software developer and holds a MSc degree in Informatics from the University of Brasilia, Brazil, where she is also an associate researcher. Her main research interests include software product lines, metaprogramming, and software development.

Leopoldo Teixeira is an Assistant Professor at the Informatics Center of the Federal University of Pernambuco, Brazil. His main research interests are in the following topics and their integration: software product lines, refactoring, and formal methods.

Demostenes Sena is an Associate Professor at Federal Institute of Education, Science and Technology of Rio Grande do Norte. He is also a PhD candidate at Federal University of Rio Grande do Norte. His main research interests are software product lines and empirical software engineering.

Uirá Kulesza is an Associate Professor at the Department of Informatics and Applied Mathematics of Federal University of Rio Grande do Norte (UFRN), Brazil. His main research interests include software product lines, generative development, and software architecture. Previously, he worked as a post-doc researcher member of the AMPLE (Aspect-Oriented Model-Driven Product Line Engineering) project.