Pós-Graduação em Ciência da Computação

Gabriela Cunha Sampaio

Partially Safe Evolution of Software Product Lines

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE
2017

Gabriela Cunha Sampaio

PARTIALLY SAFE EVOLUTION OF SOFTWARE PRODUCT LINES

Este trabalho foi apresentado à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre Profissional em Ciência da Computação.

ORIENTADOR: **Prof. Paulo Henrique Monteiro Borba**
CO-ORIENTADOR: **Prof. Leopoldo Motta Teixeira**

RECIFE
2017

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

S192p Sampaio, Gabriela Cunha
Partially safe evolution of software product lines / Gabriela Cunha Sampaio. – 2017.
107 f.: il., fig., tab.

Orientador: Paulo Henrique Monteiro Borba.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2017.
Inclui referências.

1. Engenharia de software. 2. Linhas de produtos de software. I. Borba, Paulo Henrique Monteiro (orientador). II. Título.

005.1 CDD (23. ed.) UFPE- MEI 2017-105

Gabriela Cunha Sampaio

Evolução Parcialmente Segura de Linhas de Produto de Software

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação

Aprovado em: 27/03/2017.

BANCA EXAMINADORA

_____

Prof. Dr. Paulo Henrique Monteiro Borba
Centro de Informática / UFPE
(**Orientador**)

_____

Prof. Dr. Rohit Gheyi
Departamento de Sistemas e Computação / UFCG

_____

Prof. Dr. Vander Ramos Alves
Departamento de Ciência da Computação / UnB

To my family.

Acknowledgements

À minha família, que me apoia incondicionalmente. Faço um agradecimento especial aos meus pais, Augusto e Claudia, que me motivam sempre a não desistir dos meus sonhos, e à minha irmã (e melhor amiga) Débora, que mesmo nos dias mais tristes e cansativos consegue me fazer sorrir. Obrigada por sempre me convencer de que eu sou capaz de conquistar meus objetivos.

Ao meu namorado Roberto, que é meu companheiro de todas as horas. Obrigada por aguentar meus dramas diários e por me fazer enxergar o lado bom de tudo.

Ao meu orientador Paulo Borba, que passei a admirar (ainda mais) depois de me tornar sua aluna. Sempre muito paciente comigo e presente na minha vida, tem guiado meus passos da melhor forma possível. Serei eternamente grata pelos ensinamentos. Agradeço também ao meu co-orientador Lepoldo Teixeira, que é uma inspiração para mim como profissional. Sua constante presença e apoio foram essenciais.

Aos meus amigos, que torcem tanto por mim e sempre me colocam pra cima, obrigada por tudo.

Aos membros do SPG, pelos comentários e feedbacks a respeito do meu trabalho. Em especial, gostaria de agradecer às minhas irmãs de pesquisa Klissiomara, Paola e Thaís, que se tornaram amigas. Nossa convivência e momentos de descontração são de extrema importância pra mim.

À Nicolas Dintzner, que desenvolveu uma ferramenta bastante útil no meu trabalho e estava sempre à disposição para me ajudar. Esta interação contribuiu bastante para o meu aprendizado.

À Coordenação de Aperfeiçoamento Pessoal de Nível Superior (CAPES), por ter financiado esta pesquisa.

"Mathematical reasoning may be regarded rather schematically as the exercise of a combination of two facilities, which we may call intuition and ingenuity.
(Alan Turing)

Abstract

Software product lines (SPLs) are sets of related systems that are built based on reusable artifacts. They have three elements: a variability model, that has feature declarations and dependencies among them; implementation artifacts and a configuration knowledge, that maps features to their implementation. SPLs provide several advantages, like software quality and reuse improvements, productivity gains and the capacity to customize a system depending on customers needs. There are several challenges in the SPL development context. To build customizable software and meet all customer needs, SPLs tend to increase over time. The larger a SPL becomes, the higher is the complexity to evolve it. Therefore, it is not trivial to predict which products are affected by a change, specially in large SPLs. One might need to check if products had their behaviour preserved to avoid inadvertently affecting existing users in an evolution scenario. In refactoring and conservative extension scenarios, we can avoid this problem by checking for behavior preservation, either by testing the generated products or by using formal theories. Product line refinement theories support that by requiring behavior preservation for all existing products. This happens in a number of situations, such as code refinements. For instance, in function renaming transformations, all existing products behave exactly as before the change, so we can say that this transformation is safe. Another example of SPL refinement would be changing a feature type from mandatory to optional. In this case, we increase variability, but preserving all products from the original SPL. Although several evolution scenarios are safe (or technically refinement), in many others, such as bug fixes or feature removals, there is a high chance that only some of the products are refined. In these scenarios, the existing theories would give no support, since we can not assume behaviour preservation holds for all products. To support developers in these and other non refinement situations, we define partially safe evolution for product lines, that is formalised through a theory of partial refinement that helps to precisely understand which products should not be affected by an evolution scenario. This provides a kind of impact analysis that could, for example, reduce test effort, since products not affected do not need to be tested. Additionally, we formally derive a catalog of partial refinement templates that capture evolution scenarios, and associated preconditions, not covered before. Finally, we evaluate the proposed templates by analyzing commits from two product line systems (Linux and Soletta) and we found evidence that those templates could cover a number of practical evolution scenarios.

Keywords: product line volution. product line maintenance. product line refinement

Resumo

Linhas de produto de software (LPSs) são conjuntos de sistemas relacionados desenvolvidos a partir de artefatos reusáveis. Há diversas vantagens de se trabalhar com LPS, como melhorias na qualidade do código e o aumento de reuso, e também ganhos em produtividade e uma maior customização do *software*, que se torna configurável para atender aos critérios dos clientes. Porém, há também muitos desafios. Os sistemas tendem a crescer com o tempo, o que aumenta a complexidade de evoluir a LPS. Então, a tarefa de descobrir o conjunto de produtos afetados em uma mudança se torna não trivial, principalmente em LPS maiores. Os desenvolvedores eventualmente precisam verificar se os produtos existentes preservaram comportamento para evitar afetar usuários inadvertidamente. Em cenários de refatoração e extensão conservadora, nós podemos evitar esse problema checando se o comportamento dos produtos foi preservado através da realização de testes nos produtos gerados, ou ainda com o uso de teorias formais. De fato, isso acontece em várias situações. Por exemplo, em cenários de refinamentos de código, como renomeações de funções, todos os produtos continuam se comportando exatamente da mesma forma, então nós dizemos que esta evolução é segura. Outro exemplo de refinamento de LPS seria alterar o tipo de uma *feature* mandatória para opcional. Neste caso, nós estamos aumentando variabilidade, mas preservando todos os produtos da LPS original. Apesar de haver um grande número de cenários de evolução segura (o que tecnicamente, é sinônimo de refinamento), em outros, como correções de defeitos ou remoções de *features*, existe uma chance razoável de apenas alguns produtos serem refinados. Nestes cenários, as teorias existentes não seriam capazes de dar suporte, já que nem todos os produtos preservam comportamento. Para dar suporte aos desenvolvedores nestes e em outros cenários de não refinamento, nós definimos o conceito de evolução parcialmente segura de linhas de produto de *software*, que é formalizado através de uma teoria de refinamento parcial, que ajuda a entender precisamente que produtos não devem ser afetados num cenário de evolução. Com isto, nós provemos uma espécie de análise de impacto que poderia, por exemplo, reduzir o esforço envolvido no desenvolvimento de testes, dado que produtos não afetados não precisariam ser testados. Adicionalmente, nós derivamos formalmente um catálogo de templates de refinamento parcial que capturam cenários de evolução, e pré-condições associadas, não cobertos anteriormente. Finalmente, nós avaliamos os templates propostos através de uma análise de *commits* de duas LPS (Linux e Soletta) e encontramos evidência de que os templates poderiam cobrir uma série de cenários práticos de evolução.

Palavras-chave: evolução de linhas de produto. manutenção de linhas de produto. refinamento de linhas de produto

List of Figures

List of Tables

Contents

1 INTRODUCTION

Software product lines (SPLs) provide systematic reuse and mass customization for software related products [CN01, PBvDL05]. This concept brings several advantages, such as productivity and quality improvements, apart from the capacity to customize a system based on customers needs. Nevertheless, there several challenges in the product line development field. Due to requirement changes, they naturally evolve and tend to become complex to manage. So, developers face the challenge, for example, of not inadvertently affecting users in an evolution scenario [ABKS13, PBvDL05] by guaranteeing that the evolution is safe.

This safe evolution concept [NBA+15] is formalized by a refinement notion [BTG12] that requires every product of the initial product line to have compatible behavior with at least one product of the newly evolved product line. This is useful to support developers in a number of evolution scenarios, helping them to make sure that the changes they make do not have unintended impact. For instance, users might simply need to refactor assets, or even add optional features, and these are guaranteed not to affect existing products, provided that certain conditions are observed. The refinement notion and its associated transformation templates help us to precisely capture those conditions.

Although these notions of product line safe evolution and refinement are useful in many practical evolution scenarios, they are too demanding for other scenarios because they require behaviour preservation for all products. Nevertheless, we believe that we could still support developers even when that does not apply. For example, adding functionality to an asset changes the behavior of all products that use that asset, so this is often not a product line refinement. However, the behavior of products that do not use the modified asset should not be affected. So we could still provide behavior preserving guarantees for a proper subset of the products in the product line.

This kind of partial guarantee can be useful as an impact analysis for developers to be aware of which products are affected in an evolution scenario. They could, for instance, avoid checking behavior preservation of the refined products, focusing only on testing the new functionality on the subset of products that are impacted by the changes. A notion of partially safe product line evolution could assist developers by providing this kind of weaker, but still useful, guarantee that covers common evolution scenarios not supported by refinement. This partially safe evolution concept can be helpful not only in a practical product line development context, but also in building tools that support product line development.

In fact, many evolution scenarios found in practice do not characterize a refinement.

A bug fix, or changing a top level (child of root) feature from optional to mandatory, for example, are not refinements because not all original products are refined. More specifically, in the first situation, products containing the files changed due to the bug fix are not refined; the other products, however, have the same behavior since they are not changed. When a feature is transformed from optional to mandatory, products that already had the changed feature are refined because they will present the same behavior in the new product line. However, products that did not have the feature do not preserve behavior because, in the new product line, they will present the extra behavior associated to the changed feature. Furthermore, Passos et al. [PTD+15] examined commits of the Linux repository history,[1] and found that feature removals, which are not refinements unless the feature is dead or has void behavior, often occurs. The partially safe evolution notion can address these cases by requiring refinement for a proper subset of the product line products. Transformation templates derived from this notion capture the context and required conditions for a number of scenarios, and precisely provide the subset of refined products for those cases. For example, in the feature removal scenario, the template could guarantee that products that did not have the removed feature are refined.

We formalize the partially safe evolution notion in terms of a partial refinement notion. As discussed, partially safe evolution only requires behavior preservation for a subset of the existing products in a product line.[2] For scenarios where a change is intended to refine all products, such as changing a feature from mandatory to optional, developers should rather use the refinement notion [BTG12]. Hence, they might choose to make use of the partial and refinement notions depending on the situation. Evolution in practice often interleaves different kinds of changes, ranging from refinement to no refinement scenarios. So, to support practitioners, we derive a number of properties, including that safe and partially safe evolution transformations, when applied in different orders, might lead to the same resulting product line. For example, developers could refine an asset and then remove a feature, or apply these transformations in the opposite order, and still reach the same target. In addition, we propose transformation templates representing abstractions of partial refinement situations encountered in practice. Templates work as a guide for developers. Instead of reasoning over refinement notions, they can use templates by means of pattern matching, which can also be tool supported. The partial evolution templates precisely determine which subset of products is refined for each situation; developers might even obtain this subset automatically. So our templates effectively provide change impact analysis.

To evaluate the applicability of our templates, we use the FEVER tool [DvDP16] to automatically analyze evolution scenarios found among versions 3.11 and 3.16 of the

[1] Linux repository is available at `http://github.com/torvalds/linux`.

[2] We use the “partial refinement" term to denote the new refinement notion, which requires refinement only for a subset of the original products from a product line.

Linux Kernel repository. We also analyse commits from Soletta,[3] which is a framework for making IoT devices. We found, in both projects, a number of instances of the templates in the commit history of both projects and confirm that they could have been applied, thus reinforcing the applicability of our templates. We also formalize the concepts and prove properties and soundness of the templates in the Prototype Verification System (PVS) theorem prover [OSRSC24].

To summarize, the main contributions of this work are (i) a new concept of partial product line refinement that covers partially safe evolution scenarios, (ii) a number of properties to support users not only in partially safe evolution scenarios, but also when these transformations are combined with safe evolution ones, (iii) a template catalog that represents partial safe evolution scenarios to guide developers and (iv) evidence of applicability of our templates, based on an analysis of evolution scenarios of the Linux and Soletta systems. Part of this work is already published in SPLC'16 [SBT16].

The remainder of this work is organized as follows:

- Chapter 2 provides relevant concepts for the understanding of this work;
- Chapter 3 is the core of this work. It introduces partially safe evolution and its formalisation;
- Chapter 4 proposes a template catalog, which is a possible applicability of the partial refinement theory;
- Chapter 5 provides an evaluation of the proposed templates;
- Chapter 6 discusses related work, and presents final remarks and future work.

[3] Soletta is available at `http://github.com/solettaproject/soletta`

2 BACKGROUND

In this chapter, we present relevant concepts for the understanding of this work. In Section 2.1, we define Software Product Lines (SPLs) and their basic structure using the PVS (Prototype Verification System) notation. We then introduce safe evolution of product lines in Section 2.2. Finally, we discuss the Linux Kernel notation in Section 2.3.

2.1 Software Product Lines

Software customers have different needs. For this reason, developers need to build customizable software to avoid the loss of potential customers. There are several strategies to attend customers needs. A possible approach is to develop each product separately. This might be the fastest choice, but it can increase maintainability costs, due to code clones, for example. Alternatively, product line development is a well-established mass customization and systematic reuse approach, in which large-scale products are built based on reusable artifacts. [CN01, PBvDL05]. This way, companies improve in quality aspects, since their software become well-structured and easier to maintain. Apart from this gain, productivity also increases. Following such strategy, functionalities can be represented as *features*, that are units of abstractions for both functional and non-functional requirements. They are useful to describe points of variability in product line systems. This way, each customer would choose their features and products automatically built depending on their specific needs.

For our purposes, product lines are formally represented as a triple: (1) a feature model that contains features and dependencies among them; (2) an asset mapping, that contains sets of assets and asset names; (3) a configuration knowledge, that allows features (or feature expressions) to be related to assets. In the remainder of this section, we introduce these elements in more detail.

2.1.1 Feature Models

Feature models (FMs) play a key role in SPLs. They are used to manage variability, particularly in the Feature-Oriented Domain Analysis (FODA) approach [KCH+90]. The most common notation used is a tree, whose nodes are feature names and the specific notation to specify relationships among them is observed in Figure 2.1.

In this FM, the *Mobile Media* feature is called the ***root*** feature. Every FM has at least a ***root*** feature, that is present in every product. As is described in Figure 2.1, the features with a filled circle (*Media*, *Management* and *Screen Size*) are ***mandatory***, and

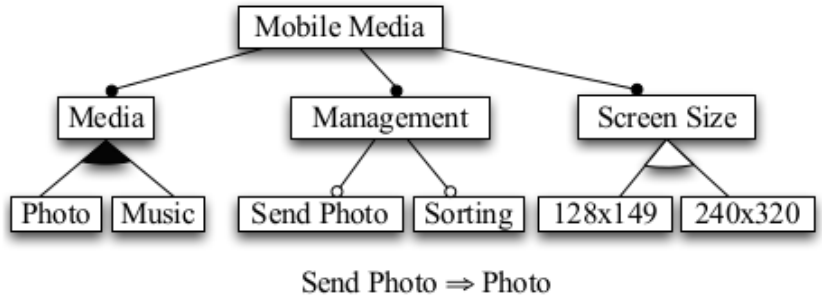

Figure 2.1 – Feature model example [NBA+15, FCS+08]

this means that if their parent is selected, they should be automatically selected. In this case, since their parent is the root feature, they must be present in every product. The *Send Photo* and *Sorting* features are classified as ***optional*** features (empty circles). So, they can be present or not in the products. However, as all the other features, they must obey the hierarchy imposed by the tree notation. Therefore, we could not select the *Send Photo* feature without selecting *Management*. Features may also be grouped in either ***or*** (filled arc) or ***alternative*** (empty arc) groups. *Photo* and *Music* compose an ***or*** group, so **at least** one of them should be selected for products containing *Media*. In contrast, for the ***alternative*** group containing features *128x149* and *240x320*, **exactly** one must be selected in products containing the *Security* feature.

Dependencies among features in a FM are also expressed through the use of cross-tree constraints (CTC). They are essentially propositional expressions involving features not directly related in the tree. In Figure 2.1, there is one example of such dependency between the *Send Photo* and *Photo* features. This CTC is needed because the tree structure is not enough to express this dependency. When interpreting such a FM, one should be aware that apart from the dependencies expressed in the tree, *Send Photo* can not be selected without *Photo*.

The product generation process begins with a feature selection, called a product configuration. There are several ways of representing configurations. For the purpose of this work, a confuguration is defined as a set of feature names, containing the features that are present in the respective product.

Feature selections can not be arbitrary; they need to obey the FM rules, as explained in the previous section. For the FM presented in Figure 2.1, these would be valid configurations:

- ✓ {Mobile Media, Media, Photo, Management, Screen Size, 128x149}
- ✓ {Mobile Media, Media, Photo, Music, Management, Send Photo, Screen Size, 240x320}

In contrast, for these configurations below, some rules were not obeyed. In the first example, the *Screen Size* feature was not selected. Since its parent, *Mobile Media* was selected and the feature is mandatory, it must also be present. In addition, rules regarding the ***alternative*** group were not obeyed, since neither *Send Photo* nor *Sorting* was selected. In the last example, more features were selected. However, from the CTC we know that *Send Photo* could not have been selected without *Photo*. Moreover, both features from the ***alternative*** group were selected and this is also not allowed.

✗ {Mobile Media, Media, Music, Management}

✗ {Mobile Media, Media, Music, Management, Send Photo, Screen Size, 128x149, 240x320}

2.1.2 Asset Mappings

When dealing with product lines, apart from FMs, we also have source code. An Asset Mapping (AM) is a set of mappings from asset names to actual assets. Each asset name is mapped to one asset only. Assets can be specified in any language, such as *Java*, *HTML* and *XML*, and have different purposes. A software project may contain test, requirements, or even configuration assets.

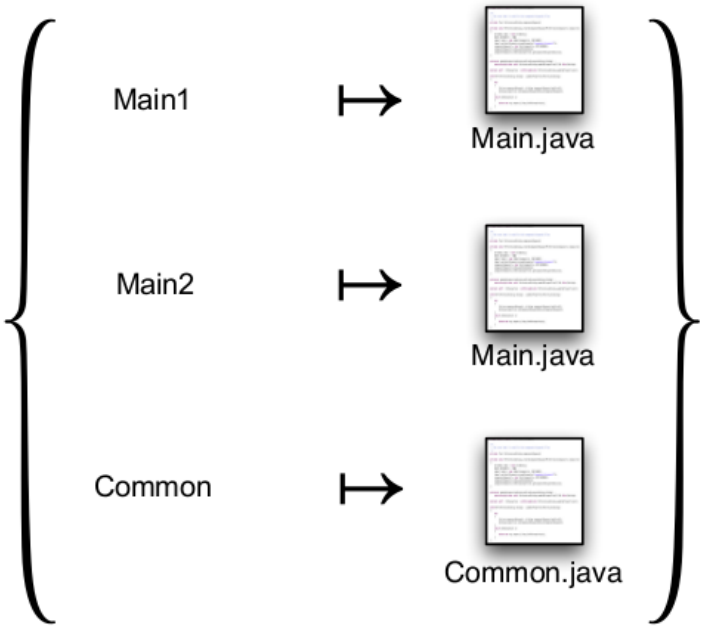

Figure 2.2 – Asset mapping example [NBA+15]

We can observe an example of an AM in Figure 2.2. On the left-hand side, there are asset names and their respective contents are mapped on the right-hand side. It is possible to notice that there are two names for the *Main.java* asset. This is due to the fact that there could be two versions of a *Main* asset, for instance, for two alternative features.

In this case, if one of the features is selected, *Main1* would be activated. Otherwise, *Main2* is included in the final product.

2.1.3 Configuration Knowledge

As already explained earlier in this section, FMs contain features and dependencies among them. The implementation is represented as an AM structure, where asset names are mapped to asset contents. Besides these two elements, it is essential to have a third structure to relate features to SPL artifacts. This is the main role of the Configuration Knowledge (CK). The CK is also essential in the products generation process. From a valid feature selection according to a FM, we process the relations from the CK to obtain the assets that implement the selected features, and consequently, the final product. The way in which we process the CK may vary according to the used notation.

Similarly to the other elements of a product line, there are several forms of specifying CKs. In some notations, they may even be implicit [Bat04, SBB+10]. In such cases, features are mapped to assets directly by their names. This only works when naming conventions are well-established in a SPL development context, like, for instance, in Ruby [TH06] projects. There is also a number of different notations for explicit CKs. In this section, we present two of them: compositional (Section 2.1.3.1) and transformation (Section 2.1.3.2) CKs.

2.1.3.1 Compositional

This notion is essentially expressed as relations between feature expressions and asset names. Here we show an example of a simple CK from the Mobile Media product line, in Figure 2.3. It is represented as a table. From this figure, we are able to locate the implementation of a determined feature or set of features. It is possible to conclude that the asset *Common.aj* is present in products containing either the *Photo* or the *Music* feature. In contrast, *AppMenu.aj* is only present in products containing the two features. Each row from the table in Figure 2.3 corresponds to an item. It is important to mention that the order of the items does not matter for this type of CK. If we permute the rows from the table in Figure 2.3, the final result is the same.

The process to generate the product for compositional CKs is straightforward. Assets are filtered according to feature expressions that match a configuration. If the expression holds, the assets are included in the final product. Otherwise, they are not included. For instance, if the configuration contains *Photo* but does not contain *Music*, the final product would have the *Common.aj* asset. *AppMenu.aj*, however, would not be included.

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

Figure 2.3 – Configuration knowledge example [NBA+15]

2.1.3.2 Transformational

Apart from compositional CKs, there are also more powerful representations. One of them is CKs associating feature expressions with transformations, instead of asset names. Several types of transformations may be applied. One example is shown in Figure 2.4. The *Mobile Media* product line is structured with the use of compositional CKs, so we created this hypothetical CK by adapting the original one, replacing asset names by transformations, just for didactic reasons.

Mobile Media	**preprocess** *MM.java, ...*
Photo	**tag** *Photo, ...*
Photo ∨ *Music*	**select** *Common.aj, ...*
Photo ∧ *Music*	**select** *AppMenu.aj, ...*
...	...

Figure 2.4 – Configuration knowledge example with transformations [NBA+15]

There are three types of transformations in Figure 2.4: **tag**, **preprocess** and **select**. For these CKs, we assume the use of preprocessing directives. So, the assets suffer a transformation before the final product is created. The use of **#ifdef** allows only a part of an asset to be the implementation of a feature, instead of the entire asset. The preprocessor then decides if the code inside the **#ifdef** is going to be included or not in the product. This depends on the **tag** being activated. In this example, this is associated to the *Photo* **tag**, that is executed when the *Photo* feature is present on the product configuration being processed.

```
class Photo {
```

```
  ...
    #ifdef Photo
      ...
    #endif
  ...
}
```

The **tag** transformation only enables the **#ifdef** with the respective name, but the code excerpt is actually included with the use of the **preprocess**, as can be seen for the feature *Mobile Media*. With the use of this transformation, the **#ifdef #endif** are removed, since they should not be part of the resulting code. Finally, the **select** transformation simply selects the asset as it is and no preprocessing is done. So, the *.aj* files are included without any change and the effect of this transformation is is the same as using a compositional CK.

In summary, this notion gives more flexibility for developers to use variability mechanisms such as *ifdefs*. Thus, the product generation process is more complex when comparing to compositional CKs, where assets do not suffer any transformation. Moreover, in this notion the order of the elements in the CK matters. For instance, when a **preprocess** comes before a **tag**, it will not make the code fragment inside the **#ifdef** to be included, since the **tag** has not been activated yet. For this reason, the final product depends on the order applied to process the items in the CK. On the following, we discuss safe evolution and how it supports developers in a SPL development context.

2.2 Product Line Refinement

Product lines naturally evolve as a consequence of changes performed to their artifacts. Supporting developers in such situations is still a challenging task. One might not know if the changes performed affected existing products in an advertent way, that would mean an impact on the products behaviour. Behaviour-preservation could be desired in several evolution scenarios, for instance, in a code refinement, such as a method addition or a function renaming. In both situations, one does not intend to affect the behaviour of existing products. Developers are often not aware if the changes performed did not inadvertently affect products. As a consequence, they might need to test the entire product line after a change.

This problem has already been tackled by establishing the safe evolution concept. A product line is safely evolved when behaviour-preservation holds for all its existing products. This idea is formalized through a refinement theory [BTG12], that has been encoded and proved using the PVS system. Thus, safe evolution technically means refinement. Before

introducing the refinement concept for product lines, it is essential to understand the formalization of its three elements.

The Prototype Verification System (PVS) [OSRSC24] provides support for the mechanization of formal specifications. This involves an integrated environment for the creation, specification, management and analysis of theories and proofs. With the use of the PVS specification language, one is able to create theories, which may include a set of axioms, lemmas, theorems, types and functions, among others.

Theories may be parameterized, and imported to other theories. PVS *uninterpeted types* are the ones that do not have a concrete definition; they are just declared but we have no information about them. When importing theories, one may instantiate *uninterpreted types* with concrete definitions. For instance, although we define the CK as an *uninterpreted type* in a higher-level theory, we then instantiate it either as a set of items or as a list of items. In every instantiation, one needs to prove that every assumptions made in the imported theory also holds for the more concrete theory. Thus, properties valid for an *uninterpreted CK type* must also be proved for its instantiations.

The specification language is based on classical, typed higher-order logic. PVS also comes with a built-in library called Prelude[1], which defines a collection of basic theories about logic, functions, predicates, sets, numbers, and other datatypes.

In this work, we use the PVS theorem prover for formalising the Partially Safe Evolution concept. Therefore, it is important to briefly review the PVS notation. So, we show how the product line refinement theory [BTG12] is encoded in PVS, as we built the partial refinement theory using its concepts.

2.2.1 Feature Models

We define FMs as abstract types. According to the PVS notation, *TYPE* means a uninterpreted type. We assume a semantics function $[\![\_]\!]$ for the FM, which yields the set of all valid configurations for a given FM. We also do not define configurations here, but they could be a set of feature names, representing the selected features by the user.

Assumption 1 (Feature Model Semantics)

$$FeatureModel : TYPE$$
$$Configuration : TYPE$$
$$[\![\_]\!] : FeatureModel \to \mathcal{P}[Configuration]$$

With the semantics notion, we are able to reason over FM equivalence and refinement. Two FMs are equivalent, as stated in Definition 1, when they have the same semantics.

[1] http://www.cs.rug.nl/~grl/ar06/prelude.html

Definition 1 (Feature Model Equivalence)

$$\cong (fm1, fm2) : bool = [\![fm1]\!] = [\![fm2]\!]$$

We define FM refinement in Definition 2. A FM $fm2$ refines a FM $fm1$ when the latter semantics is a subset of the former semantics. So, refinement in the context of FMs means preserving all existing configurations. We may have possibly new ones, but we should be able to still generate the initial ones. This happens when, for example, we add an optional feature to a FM. We still have the initial configurations, but we are able to generate new ones with the added feature.

Definition 2 (Feature Model Refinement)

$$\sqsubseteq (fm1, fm2) : bool = subset?([\![fm1]\!], [\![fm2]\!])$$

We will see later on this chapter that product line refinement does not lead to feature model refinement. For example, feature renamings are safe evolution scenarios, and consequently product line refinements, as we do not change the behaviour of any product; only a feature name is changed. However, according to Definition 2, this type of scenario is not considered a FM refinement in our concrete FM notions. When we change a feature name, we are possibly changing configurations, that might be defined as sets of feature names. So, a number of initial configurations may not be present in the evolved FM semantics, which means non-refinement.

2.2.2 Asset Mappings

We also rely on a definition of assets. Similarly to the FM, $Asset$ is also an abstract type. We also assume a refinement function that returns true when the two sets of assets have compatible behavior. For instance, an asset could be modified with a function renaming operation. The observable behavior of both assets would be the same, so $\sqsubseteq$ holds for the set containing the initial asset and the set containing its modified version, respectively. The wf function takes a set of assets, that represent a product, and informs whether the product is well-formed. This function is not defined, since there could be several interpretations of well-formedness. For example, one could argue that a product is well-formed if it is compiling successfully. Alternatively, a valid product could mean that the program is not raising exceptions to its final user. For this reason, we assume that there might be a way to determine if a program is well-formed, but we do not present a definition for it.

Assumption 2 (Asset refinement)

$$
\begin{aligned}
&Asset : TYPE \\
&\sqsubseteq : \mathcal{P}[Asset], \mathcal{P}[Asset] \rightarrow bool \\
&\text{wf} : \mathcal{P}[Asset] \rightarrow bool
\end{aligned}
$$

We assume that the refinement relation $\sqsubseteq$ is a preorder. Reflexivity is essential here, and this is aligned with the idea that refinement means "equal or better". Consequently, every set of assets needs to refine itself. The fact that two sets of assets are equal imply that they have the same observable behavior. Thus, it is considered a refinement. As shown in Axiom 1, transitivity also holds for any set of assets. If a set of assets as is refined by a set of assets bs and bs is refined by the set cs, cs also refines as.

Axiom 1 (Asset set refinement is a preorder)

$$
\begin{aligned}
&\forall as : \mathcal{P}[Asset] \cdot as \sqsubseteq as \\
&\forall (as, bs, cs : \mathcal{P}[Asset]) \cdot as \sqsubseteq bs \wedge bs \sqsubseteq cs \Rightarrow as \sqsubseteq cs
\end{aligned}
$$

As we explain in Section 2.1.2, AMs are structures that allow the correlation between asset names and assets. We rely on a definition of asset names as an uninterpreted type, as described in Assumption 3. Then, we define the AM type as a mapping between asset names and assets. Technically, a set is a mapping when asset names map to a single asset. This is also specified in Assumption 3.

Assumption 3 (Asset mappings)

$$
\begin{aligned}
&AssetName : TYPE \\
&mapping(r : \mathcal{F}[AssetName, Asset]) : bool = \\
&\qquad \forall n : AssetName \cdot \forall a, b : Asset \cdot (n, a) \in r \wedge (n, b) \in r \Rightarrow a = b \\
&AM : \{r : \mathcal{F}[AssetName, Asset] \mid mapping(r)\}
\end{aligned}
$$

Similarly to the FM, we also have a refinement notion for AMs. First, we require that the AMs have the same domain. Additionally, for every asset found in the initial AM $am1$, there needs to be an asset in the evolved AM $am2$ with the same name, that refines the initial one. AM refinement holds in several situations, like in a function renaming scenario. If we rename a function in each initial asset, all of them are refined by the new ones.

Definition 3 (Asset mapping refinement)

$$\begin{aligned}\sqsubseteq (am1, am2) : bool = \\ & (dom(am1) = dom(am2) \wedge \\ & (\forall an \in dom(am1) \cdot \\ & \quad \exists a1, a2 \cdot (an, a1) \in am1 \wedge (an, a2) \in am2 \wedge a1 \sqsubseteq a2))\end{aligned}$$

Nevertheless, some scenarios do not represent AM refinement. For example, adding new assets to an AM is not a refinement, since we are changing the domain in this case, so the first condition in Definition 3 does not hold. Furthermore, asset name renamings are also not AM refinements. Although these would not have any impact to the assets behaviour, at least one of the initial assets would be refined by an asset with a different name, so instead of having $(n, a1)$ in the initial AM and $(n, a2)$ in the final AM, given that $a1 \sqsubseteq a2$ we would have $(n, a1)$ and $(n', a2)$, respectively.

2.2.3 Configuration knowledge

As already mentioned in Section 2.1.3, the CK can have several representations, like, for example, the compositional (Section 2.1.3.1) and transformations (Section 2.1.3.2) CK. So, the CK is represented as an uninterpreted type. Similarly to the FM, there is also a semantics function for the CK, which yields the product given a CK, an asset mapping and a configuration. We define products as finite sets of assets. Depending on the language used, the semantics function also may vary. For this reason, we do not give an interpretation of semantics here.

Assumption 4 (Configuration knowledge Semantics)

$$\begin{aligned}& CK : TYPE \\ & [\![\_]\!] : CK \rightarrow AM \rightarrow Configuration \rightarrow \mathcal{F}[Asset]\end{aligned}$$

Like the FM, with the CK semantics function we are able reason over CK equivalence. As stated in Definition 4, two CKs are equivalent whenever they have the same semantics. This means that the CK evaluation needs to be equal considering any AM and configuration. As we may consider configurations from a specific FM, we also have a weaker equivalence notion (Definition 5), which is akin to Definition 4, since it requires the CK evaluation to be equal only according to configurations from a specific FM.

Definition 4 (Configuration Knowledge Equivalence)

$$\cong (ck1, ck2) : bool = [\![ck1]\!] = [\![ck2]\!]$$

Definition 5 (Configuration Knowledge Weaker Equivalence)

$$\begin{aligned}\cong_F (ck1, ck2) : bool = \\ \forall A \cdot \forall c \in F \cdot [\![ck1]\!]^A_c = [\![ck2^A_c]\!]\end{aligned}$$

All relations presented here are pre-orders, which means that they are reflexive and transitive. It is not intuitive to think in a refinement or equivalence notion that is not a pre-order, as this allows they to be valid considering the same element and after refining an element several times, we still need to provide support through transitivity theorems. We do not present such properties here, but they can be found in previous work [BTG12].

2.2.4 Software Product Line

A software product line is a well-formed triple formed by a FM, an AM and a CK. We assume that a product line is well-formed, as stated in Definition 6, whenever all valid products are well-formed. We do not have a concrete definition of product well-formedness, since this would require us to adopt a particular programming language. So, it can mean that the product is compiling without errors, for example.

Definition 6 (Software product line)
For a feature model F, an asset mapping A, and a configuration knowledge K, we say that the tuple (F, A, K) is a product line when, for all $c \in [\![F]\!], wf([\![K]\!]_c^A)$

These concepts just discussed are useful to introduce the notion of safe evolution of product line, that is formalized through a refinement relation. Definition 7 establishes that for any product lines L and L', the latter refines the former when for each configuration c from the initial product line there is a configuration c' in L' so that the product generated from c in L is refined by the one generated from c' in L'. Therefore, the central idea is that the resulting product line might even have more products than before, possibly implementing new features, as long as current users are not affected, i.e., all existing products must be refined by new ones.

Definition 7 (Product line refinement)
For arbitrary product lines $L = (F, A, K)$ and $L' = (F', A', K')$, L' refines L, denoted by $L \sqsubseteq L'$, whenever

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

Product line refinement holds for several evolution scenarios. For instance, when one adds an optional feature being able to generate more products than before and preserving the behaviour of existing products, this is considered to be a refinement. Apart from this scenario, transforming a feature type from mandatory to optional is also safe evolution, since the evolved product line has more products than the original one, but behaviour is preserved for all existing products.

It is important to notice that according to Definition 7, feature names do not matter. The reason for this is that the refined product can be originated from a configuration c'

different from the configuration of the original product (c). As a consequence, changing a feature name is also a product line refinement.

We now relate AM refinement to product refinement. As described in Axiom 2, if an AM $am1$ is refined by an AM $am2$, refinement also holds for products generated based on both AMs.

Axiom 2 (AM refinement implies product refinement)

$$\begin{aligned}&\forall(am1, am2) : am1 \sqsubseteq am2 \Rightarrow \\ &\quad \forall(K : CK, c : Conf) : wf([\![K]\!]_c^{am1}) \Rightarrow \\ &\quad wf([\![K]\!]_c^{am2}) \wedge [\![K]\!]_c^{am1} \sqsubseteq [\![K]\!]_c^{am2}\end{aligned}$$

2.2.5 Compositionality

When working in product line development contexts, developers may need to change only one of the three elements in an evolution scenario. For instance, one could change only a source code artifact without touching the FM and the CK, or alternatively only add a feature to the FM. The product line refinement theory also provides compositionality theorems to support developers in such situations. So, we can be sure that refining a single element implies refining the entire product line.

As we can see in Theorem 1, one can change a FM F from an initial product line (F, A, K) obtaining a new FM F'. So, if this change represent a FM refinement, and the new product line is well-formed, the evolved product line formed by the new FM and the existing AM and CK (F', A, K) refines the initial product line. Thus, FM refinement leads to product line refinement.

Theorem 1 (Feature Model Refinement Compositionality)
For product lines $L = (F, A, K)$ and $L' = (F', A, K)$, if $F \sqsubseteq F'$ and L' is well-formed, then $L \sqsubseteq L'$.

Nevertheless, as we mentioned earlier in this section, product line refinement does not imply FM refinement. This happens in every situation where we preserve existing products but do not preserve existing configurations. In feature renaming scenarios, for example, all products have their behaviour preserved, which means product line refinement. However, configurations possibly change, since we are replacing feature names. So, we do not generate the existing configurations in the new FM, so this is not a FM refinement. This situation can also happen when developers remove features which have void behaviour. In this scenario, we existing users are not affected, as the removed feature does not present any extra behaviour. So, this characterizes a product line refinement. However, it is also not a FM refinement, as we are not able to generate configurations with the removed feature in the evolved FM.

As Theorem 2 shows, AM refinement also leads to product line refinement. This is intuitive, since AM refinement requires assets to be refined, we still preserve product behaviour. Moreover, we can also guarantee that the existing product line remains well-formed, as we only perform asset refinements.

Theorem 2 (Asset Mapping Refinement Compositionality)
For product lines $L = (F, A, K)$ and $L' = (F, A', K)$, if $A \sqsubseteq A'$, then $L \sqsubseteq L'$ and L' is well-formed.

In Definition 5, we introduce CK weaker equivalence. One might need to guarantee that changes to the CK lead to product line refinement. So, as stated in Theorem 3, if a CK K is equivalent to a CK K' according to configurations from the initial FM F, the resulting product line L', which is formed by F, A, and K', refines the initial one.

Theorem 3 (Configuration Knowledge Weaker Equivalence Compositionality)
For product lines $L = (F, A, K)$ and $L' = (F, A, K')$, if $K \cong_F K'$, then $L \sqsubseteq L'$ and L' is well-formed.

2.3 Linux Overview

Linux[2] is a highly configurable system that runs in several devices (mobile phones, Desktops, printers, among others) and has more than 10,000 features and 6 million lines of C code.

In summary, the Linux system has three main elements. Its variability model is expressed in terms of the Kconfig language. In Kconfig, it is possible to declare features and dependencies among them. The configuration knowledge is the set of mappings obtained in Makefiles. Finally, the third part is the implementation. This is analogous to the asset mapping already introduced in Section 2.1.2, since we have assets and their respective names. In the remainder of this section, we present each element in more detail.

Kconfig

A Kconfig excerpt example can be found in Listing 2.1. Every declaration in Kconfig starts with the `config` word. Then, the feature name is specified, which is `EXYNOS_AUDSS_CLK_CON` in this case. Its short description informs that this is a Samsung driver for clock controller support. Features can be of type bool, assuming "Y" or "N" values depending if they have been selected. Tristate features can assume boolean values and also "M", which means that it is present as a dinamically loadable kernel module.

[2] Linux is available at `http://github.com/torvalds/linux`

In Kconfig, it is also possible to establish cross-tree constraints among features. Developers may use `depends on` to indicate features that must be activated before, so `EXYNOS_AUDSS_CLK_CON` can not be selected if `COMMON_CLK_SAMSUNG` is not selected. In other declarations we may find the `select` keyword to indicate features that are automatically selected when the current feature is selected. So, it is a dependency in the opposite direction to `depends on`. One may set default values as well. In this case, if `ARCH_EXYNOS` is selected, the `EXYNOS_AUDSS_CLK_CON` feature assumes "Y" by default. Otherwise, it has no default value. The text that comes after the `help` word is exhibited when users need more information about the feature.

Listing 2.1 – Excerpt of "linux/drivers/clk/samsung/Kconfig"

```
config EXYNOS_AUDSS_CLK_CON
        tristate "Samsung Exynos AUDSS clock controller support"
        depends on COMMON_CLK_SAMSUNG
        default y if ARCH_EXYNOS
        help
          Support for the Audio Subsystem CLKCON clock controller
          present on some Exynos SoC variants. Choose M or Y here if
          you want to use audio devices such as I2S, PCM, etc.
```

The Kconfig notation is different from the tree notation introduced in Section 3.2.2.1. For example, features in Kconfig are not classified into *optional*, *mandatory*, *alternative* and *or*. Every dependency among features is expressed through the *depends on* and *select* statements. So, one could argue that if users are able to select or deselect a feature, it might be optional. Mandatory features often must appear in every product. If two features can not be selected at the same time, but exactly one of them should be selected, we could classify them into *alternatives*. A similar procedure could be applied to *or* features. Nevertheless, this is a challenging task and we are not aware of any tool that classifies Linux features into these four categories automatically.

Makefiles

The CK is expressed through Makefile mappings in the Linux system. There are several Makefiles and one might import mappings from another. In these files, features are mapped to assets. So, the idea is basically the same of the CK notion previously presented in Section 2.1.3. The Makefile presented in Listing 2.2 has some of the mappings related to Samsung driver features, and one of them is the mapping of the feature presented in the Kconfig notation (*EXYNOS_AUDSS_CLK_CON*). As we can see, it is mapped to the *clk-exynos-audss.o* artifact. In the implementation, we will possibly find a file with the same name, but with the *.c* extension instead of *.o*. This mapping does not necessarily

mean that this artifact implements exclusively the mapped feature. As we discussed in Section 2.1.3.2, features may also be implemented with the use of **#ifdefs** blocks. So, a single artifact may implement more than one feature. Pre-processing steps guarantee that products are only generated with the implementation of the selected features and nothing else.

Listing 2.2 – Excerpt of "linux/drivers/clk/samsung/Makefile"

```
obj-$(CONFIG_SOC_EXYNOS5260) += clk-exynos5260.o
obj-$(CONFIG_SOC_EXYNOS5410) += clk-exynos5410.o
obj-$(CONFIG_SOC_EXYNOS5420) += clk-exynos5420.o
obj-$(CONFIG_EXYNOS_ARM64_COMMON_CLK) += clk-exynos5433.o
obj-$(CONFIG_SOC_EXYNOS5440) += clk-exynos5440.o
obj-$(CONFIG_EXYNOS_AUDSS_CLK_CON) += clk-exynos-audss.o
obj-$(CONFIG_ARCH_EXYNOS) += clk-exynos-clkout.o
obj-$(CONFIG_EXYNOS_ARM64_COMMON_CLK) += clk-exynos7.o
obj-$(CONFIG_S3C2410_COMMON_CLK) += clk-s3c2410.o
obj-$(CONFIG_S3C2410_COMMON_DCLK) += clk-s3c2410-dclk.o
```

Implementation

The main language used in the Linux system is *C*. As we are using the *EXYNOS_AUDSS_CLK_CON* feature as example, we present part of its implementation in Listing 2.3. We should observe that this code excerpt is inside an *ifdef* block. So, it is only included if the *CONFIG_PM_SLEEP* feature is included. The rest of the code, that is not presented here but is available on the GitHub repository,[3] is not inside any *ifdef*, so it is included for every feature mapped to this artifact in Makefiles.

Listing 2.3 – Excerpt of "linux/drivers/clk/samsung/clk-exynos-audss.c"

```
#ifdef CONFIG_PM_SLEEP
static unsigned long reg_save[][2] =
{...};

static int exynos_audss_clk_suspend(struct device *dev)
{...}

static int exynos_audss_clk_resume(struct device *dev)
{...}
#endif /* CONFIG_PM_SLEEP */
```

[3] http://github.com/torvalds/linux/blob/master/drivers/clk/samsung/clk-exynos-audss.c

Product Generation Process

Now, we give an overview of the product generation process for the Linux Kernel system in Figure 2.5. On the left end of the diagram, we have Kconfig models, Makefiles and the Implementation structures. First, the Kconfig models are rendered in a configurator, which is enabled, for instance, by the execution of the *make menuconfig* command in the Linux terminal. Then, the user can see which configurations are available and make a valid selection. A *.config* file is generated representing the user selection. Features selected have the "Y" value, and the others have "N" or do not appear.

After these three initial steps, we have a valid configuration. In the fourth step, the user runs the *make* command, which initiates the execution of the top Makefile at the root of the Linux source code tree. So, in Step 5.1, the top Makefile invokes *config* to read the *.config* file. This file is then translated into two other files in Step 5.2. The *auto.conf* is useful for *make*, and *autoconf.h* is later used by the C pre-processor *cpp*.

So, depending on the features selected in the *.config* file, assets are included in the final product. The top Makefile controls the resident kernel image vmlinux and the kernel loadable modules. To build vmlinux, Kbuild first builds all the object files stored in *core-y*, *libs-y*, *drivers-y*, and *net-y* variables. For example, the *EXYNOS_AUDSS_CLK_CON* feature shown would be present in the drivers list if selected by the user. To generate object files, Kbuild compiles *.c* files with equal name to the *.o* files present in Makefiles. In Step 5.3, Kbuild adds an inclusion directive to *autoconf.h* in each target source file. This header file contains macro definitions for the features selected during configuration. It is encoded as follows: all features in the *.config* file result in preprocessor symbols with the same name; tristate features selected as modules are suffixed with _MODULE; macros of selected Boolean/tristate features are set to 1; integer/string features, if present, lead to macros whose values match those given during configuration.

Then, all *ifdefs* are evaluated by the pre-processor and code blocks are included or not depending on the features selected. Finally, the generated code is compiled by the C compiler and the object files are linked and merged into a single *built-in.o* file, which is then linked into vmlinux by the parent Makefile. Similarly, tristate features set to "M", after linkage, result in loadable kernel objects (.ko files).

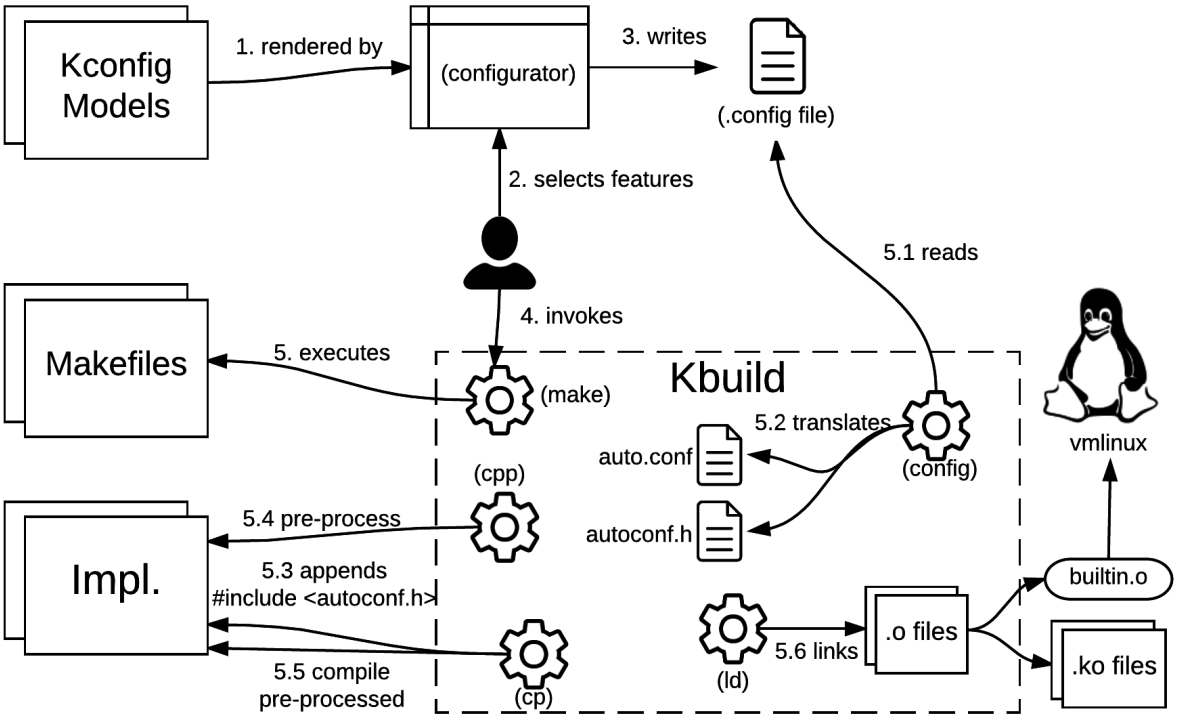

Figure 2.5 – Linux Structure Overview (adapted from [PTD+15])

3 PARTIALLY SAFE EVOLUTION

As we discuss in Chapter 2, the product line refinement theory [BTG12] could support developers in several situations, such as refactoring a particular class or changing a feature from mandatory to optional. Nevertheless, developers might need to perform *unsafe* changes, for example, when fixing bugs. These scenarios are not entirely safe because it is desirable that at least some products do not have their behaviour preserved. However, the evolution might be safe according to a subset of product configurations. Therefore, we call them *partially safe*.

The safe evolution concept is not able to support developers in partially safe scenarios, since it only provides support for cases where all products are refined. In this chapter, we introduce and formally establish the concept of partially safe evolution for product lines. First, in Section 3.1, we present a motivating example taken from the Linux Kernel evolution history. Then, we formalize partially safe evolution through a partial refinement theory, in Section 3.2. We also explore properties and how partial refinement relates to refinement. In Section 3.3, we discuss final remarks about our approach.

3.1 Motivating Example

To illustrate a common evolution scenario not covered by the product line refinement notion, we refer to commit *ae3e4c2776*[1] of the Linux repository history. It basically consists of a feature removal scenario. The *LEDS_RENESAS_TPU* feature represents a LED driver in the Linux system; this feature was removed because it was superseded by the preexisting generic *PWM_RENESAS_TPU* driver. The commit changes are illustrated in Listings 3.1, 3.2 and 3.3. We use the "−" symbol in each line to indicate that it was removed from the file.

In Listing 3.1, we observe changes to a Linux Kconfig file,[2] which, as discussed in Section 2.3, models features and their properties, and plays a similar role to variability models such as feature models. As already explained, statements in Kconfig declare features by indicating their names, types (in this case, a *boolean*, so it can assume y or n, depending on its selection) and relations with other features. In this example, *LEDS_RENESAS_TPU* depends on *LEDS_CLASS*, *HAVE_CLK* and *GPIOLIB*. Thus, the former can only be selected if these three other features are also selected in the product. In terms of feature

[1] Feature removal commit http://github.com/torvalds/linux/commit/ae3e4c2776. Laurent Pinchart commited on Jul 16, 2013; version v3.12-rc1.

[2] Kconfig language documentation https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt.

models, this condition is akin to establishing *LEDS_RENESAS_TPU* as a descendant of those features.

Listing 3.1 – Changes made to "drivers/leds/Kconfig"

```
-config LEDS_RENESAS_TPU
-    bool "LED support for Renesas TPU"
-    depends on LEDS_CLASS=y && HAVE_CLK && GPIOLIB
-    help
-    ...
```

The *LEDS_RENESAS_TPU* feature is implemented by the *leds-renesas-tpu.o* asset, as we can see in the makefile partially presented in Listing 3.2. These files represent Linux configuration knowledge, relating feature expressions (presence conditions) to asset names. This mapping was removed, since the intention was to remove the feature. However, a feature is only completely removed when its implementation is deleted as well, otherwise there would be unused assets. Listing 3.3 indicates that this was actually done; we only show part of the code associated to *LEDS_RENESAS_TPU*, but the *leds-renesas-tpu.c* and *leds-renesas-tpu.h* files were entirely removed from the repository.

Listing 3.2 – Changes made to "drivers/leds/Makefile "

```
-obj-$(CONFIG_LEDS_RENESAS_TPU) += leds-renesas-tpu.o
```

Listing 3.3 – Changes made to "drivers/leds/leds-renesas-tpu.c"

```
-#include <linux/module.h>
-#include <linux/init.h>
- ...
-MODULE_LICENSE("GPL v2");
```

After these changes, the new product line likely will not have products with behavior compatible to products that had feature *LEDS_RENESAS_TPU* in the original product line. Unless *LEDS_RENESAS_TPU* had a void behavior, or side effect free behavior equivalent to another feature, the evolution will not be safe for products containing *LEDS_RENESAS_TPU* in their configuration, unless the *PWM_RENESAS_TPU* feature has a compatible behavior with the previous one and the products having the former also had the latter, but this may not be true. Thus, in the resulting product line, we likely do not find products that match the behavior of original products with *LEDS_RENESAS_TPU*. Consequently, this is not considered a safe evolution scenario; the existing theory fails to support developers in this case, even though we know that products that do not have that feature should have the same behavior.

In fact, this scenario is partially safe considering the product configurations that do not select *LEDS_RENESAS_TPU* and thus are not impacted by its removal. Since Linux users can optionally select *LEDS_RENESAS_TPU*, there might be a number of products that do not have it. Supposing that 50% of the products select *LEDS_RENESAS_TPU*, we could give guarantee behaviour preservation for half of the products, significantly benefiting developers, for instance, in productivity, by avoiding these products to be tested. Moreover, a partially safe evolution concept could also avoid developers to unintentionally affect products. For instance, in this feature removal scenario, products not containing *LEDS_RENESAS_TPU* should not be affected. So, developers would be aware that artifacts not implementing the removed feature should not be affected. If, by a mistake, the set of products refined is not the set expected, developers would probably check their changes and discover possible mistakes.

There are many other kinds of partially safe evolution scenarios, such as asset additions, where both implementation files and the respective mappings are added to the product line. In this scenario, products that include the new files likely do not preserve behavior, but the evolution is still safe when considering products that do not have the added files. The percentage of refined products tends to increase when the frequency of the features in configurations implemented by the added assets decreases. If the affected feature is mandatory, guarantee is limited, since this feature possibly appears in all products (in this case we would give no guarantee). In contrast, when the changed feature is optional and directly connected to the root feature, we could guarantee, depending on the situation, for example, 50% of the products and this percentage tends to increase when the feature is positioned lower in the tree. Therefore, we believe that one could benefit from partially safe evolution notion, that is able to handle unsafe evolution scenarios, while still offering safe evolution guarantees considering a subset of the products.

3.2 Partial Refinement

To handle evolution scenarios such as the one illustrated in the motivating example, we introduce a theory of partial refinement that formalizes our notion of partially safe evolution of product lines. Moreover, we also present some properties derived from our partial refinement definition. Additionaly, we analyze how refinement and partial refinement operations can be interleaved, which might be often necessary in practice. Although we specify our theory in PVS, we do not use its syntax here to improve readability.

The partial notion assumes that only some products are refined. So, in Definition 8, we use S as an index to denote the subset of refined product configurations. More precisely, for product lines L and L', and set of configurations S, we say that L' partially refines L with respect to S when product configurations from S are valid for both FMs, and product

refinement holds for all such configurations. The first condition is necessary to guarantee that all configurations in S are valid according to the respective product lines. Otherwise, we would not be able to generate valid products.

Definition 8 (Partial product line refinement)
For arbitrary product lines $L = (F, A, K)$ and $L' = (F', A', K')$, and a set of configurations S, L' partially refines L for the configurations in S, denoted by $L \sqsubseteq_S L'$, whenever

$$S \subseteq [\![F]\!] \ \wedge \ S \subseteq [\![F']\!] \ \wedge \ \forall c \in S \ \cdot \ [\![K]\!]_c^A \sqsubseteq [\![K']\!]_c^{A'}.$$

With this relation, we can support developers in examples like the one in Section 3.1. We could simply associate L with the product line before the feature removal, and L' represents the resulting product line after removing *LEDS_RENESAS_TPU*. Thus, S would be the set of all configurations that do not contain *LEDS_RENESAS_TPU*. This would include configurations such as $\{IMX\_WEIM, MVEBU\_MBUS, OMAP\_OCP2SCP, ...\}$ and $\{ADB\_IOP, ADB\_MACII, PROC\_EVENTS, ...\}$. Since the only modification is the feature removal, and we filter the respective changed products by verifying refinement only for configurations in S, partial refinement holds. Partial refinement would not hold, for example, if S included configurations containing *LEDS_RENESAS_TPU*. Hence, considering that we give guarantees that the other products are refined, developers would only need to test products that had *LEDS_RENESAS_TPU*, which could consequently increase productivity. The previous theory gives no guarantees for this case, so developers would have no support.

The partial refinement relation is reflexive and transitive, which are essential conditions to support stepwise partially safe evolution. Theorem 4 establishes that every product line is partially refined by itself. As required by Definition 8, we need to assure that S is a subset of the valid configurations generated from the respective product line.

Theorem 4 (Partial product line refinement reflexivity)
For an arbitrary product line $L = (F, A, K)$, and a set of configurations S, if $S \subseteq [\![F]\!]$, then $L \sqsubseteq_S L$.

Proof. Let $L = (F, A, K)$ be an arbitrary product line. By Definition 8, we have to prove that $S \subseteq [\![F]\!]$ and $\forall c \in S \cdot [\![K]\!]_c^A \sqsubseteq [\![K]\!]_c^A$. The first condition is already assumed by the theorem and the second follows from asset refinement reflexivity (see Axiom 1). □

One might want to consecutively perform partial refinement operations, and the transitivity property guarantees that this is feasible, and that it results in refined products. However, given that the consecutive partial refinement operations might involve different subsets of products, we can only guarantee that refinement holds for the intersection of

the configurations refined in each step. For instance, given a product line L_1, one could first fix a bug, resulting in a product line L_2, and then remove a feature, obtaining L_3. Assuming that S and T are the sets of configurations refined in each step, S would be the set of configurations whose products do not contain the changed files in the bug fix, and T would be the set of configurations that do not have the removed feature. Assuming that S and T are different, the resulting product line L_3 does not partially refines L_1 in terms of S or T in isolation, because the products refined in the first step are not necessarily refined in the second step, and vice versa. But L_3 partially refines L_1 for the configurations that are in both sets: $S \cap T$.

Theorem 5 (Partial product line refinement transitivity)
For arbitrary product lines L_1, L_2, L_3, and set of configurations S and T, if $L_1 \sqsubseteq_S L_2$ and $L_2 \sqsubseteq_T L_3$, then $L_1 \sqsubseteq_{S \cap T} L_3$.

Proof. Let $L_1 = (F_1, A_1, K_1)$, $L_2 = (F_2, A_2, K_2)$ and $L_3 = (F_3, A_3, K_3)$ be arbitrary product lines. Assume that $L_1 \sqsubseteq_S L_2 \wedge L_2 \sqsubseteq_T L_3$. By Definition 8, this amounts to:

$$S \subseteq [\![F_1]\!] \wedge S \subseteq [\![F_2]\!] \tag{3.1}$$

$$\forall c \in S \cdot [\![K_1]\!]_c^{A_1} \sqsubseteq [\![K_2]\!]_c^{A_2} \tag{3.2}$$

$$T \subseteq [\![F_2]\!] \wedge T \subseteq [\![F_3]\!] \tag{3.3}$$

$$\forall c \in T \cdot [\![K_2]\!]_c^{A_2} \sqsubseteq [\![K_3]\!]_c^{A_3} \tag{3.4}$$

We then have to prove that

$$(S \cap T) \subseteq [\![F_1]\!] \wedge (S \cap T) \subseteq [\![F_3]\!] \tag{3.5}$$

and

$$\forall c \in S \cap T \cdot [\![K_1]\!]_c^{A_1} \sqsubseteq [\![K_3]\!]_c^{A_3} \tag{3.6}$$

We can trivially prove Predicate 3.5 by using Predicate 3.1 and Predicate 3.3. To prove Predicate 3.6, assuming an arbitrary $c \in S \cap T$, we have to prove that $[\![K_1]\!]_c^{A_1} \sqsubseteq [\![K_3]\!]_c^{A_3}$. Properly instantiating c in Predicate 3.2 and Predicate 3.4, we have that $[\![K_1]\!]_c^{A_1} \sqsubseteq [\![K_2]\!]_c^{A_2}$ and $[\![K_2]\!]_c^{A_2} \sqsubseteq [\![K_3]\!]_c^{A_3}$. The proof then follows by asset set refinement transitivity (see Axiom 1). □

Note that S and T might be disjoint. For example, let us consider, for simplicity, a product line with two features A and B only. One could first remove feature A and then feature B. S in this case would be the set of product configurations that do not have A, that is the configuration that has just the feature B. In the second transformation, however, T corresponds to product configurations that do not have B, that is the product configuration that have A only. So, S and T are disjoint in this case, since they do not

share any configuration. So, we would not be able to give guarantee for any product. This may happen in a number of scenarios and we can not avoid this situation, but for others we still give support after several transformations, specially considering a system like the Linux Kernel that has thousands of features, it is expected that for most of the possible evolution scenarios, one could still be supported after performing consecutive partially safe transformations.

3.2.1 Considerations

To analyse implications of the partial refinement definition, we now discuss two important aspects of our definition.

3.2.1.1 Feature names matter

We should observe that in Definition 8 the configuration c remains the same for both the original and refined product lines. Thus, we are not giving guarantees for any refined product that has a different configuration from the initial one. Consequently, partial refinement only holds for products containing features whose names remain the same. As we have previously mentioned in Section 2.1, we define configurations as sets of feature names. So, a change in a feature name affects configurations containing that name. For this reason, feature names matter for partial refinement. We can still give guarantees for the other products that are not affected (in this case, products not containing the renamed feature). However, we would not be able to give the full guarantee desired according to Definition 8. So, the refinement notion is more useful in this case, as it gives guarantees for all products.

A scenario that involves only a change to a feature name would not be a partial refinement, according to how we formalize it in Definition 8. For instance, consider the commit 84743*ea*369 [3] extracted from the Linux repository history. The following listings represent the changes performed in this commit. The lines starting with the + and − symbols were added and removed, respectively.

Listing 3.4 – Changes made to “drivers/gpio/Kconfig ”

```
-config GPIO_LANGWELL
-bool "Intel Langwell/Penwell GPIO support"
+config GPIO_INTEL_MID
+       bool "Intel Mid GPIO support"
```

Listing 3.5 – Changes made to “drivers/gpio/Makefile”

```
-obj-$(CONFIG_GPIO_LANGWELL)      += gpio-langwell.o
```

[3] https://github.com/torvalds/linux/commit/84743ea369

```
+obj-$(CONFIG_GPIO_INTEL_MID)    += gpio-intel-mid.o
```

Listing 3.6 – Changes made to "drivers/gpio/gpio-langwell.c → drivers/gpio/gpio-intel-mid.c "

```
File renamed without changes.
```

In this commit, the three parts of the Linux product line are changed. However, all changes are related to modifying the feature name. The $GPIO\_LANGWELL$ feature was renamed to $GPIO\_INTEL\_MID$. Thus, the feature name is changed in the Kconfig in Listing 3.4. Moreover, a Makefile mapping also needs to be updated, as shown in Listing 3.5. Finally, the source file name has also been modified, as Listing 3.6 shows. After such changes, configurations containing $GPIO\_LANGWELL$ now have $GPIO\_INTEL\_MID$ instead.

This scenario could be considered a refinement, since only the feature name has been changed and all existing products have their behaviour preserved after the change. Intuitively one would classify it as a partial refinement according to all configurations. Nevertheless, as stated in Definition 8, we require the evolved product to have the same configuration of the original one. Therefore, products whose configurations were affected would not be considered to be refined according to our definition for partial refinement.

To handle feature renaming scenarios, we have an alternative (and more elaborate) definition for partial refinement, as we discuss in Section 3.2.3. We use Definition 8 as default for simplicity and avoid confusion, since it is able to deal with any other scenario except feature renamings.

3.2.1.2 Could S be the emptyset?

As already explained earlier in this section, we define partial refinement as an indexed relation. The index is a set of configurations S. It is important to notice that according to Definition 8, we only check refinement for products in the scope of S. This means that the larger the S, the higher is the number of refined products and stronger the provided guarantees. Consequently, developers can be better supported, since they would be aware that there is a significant number of products that are refined, which would avoid the need for testing.

Nevertheless, developers can always choose a subset of S, given that partial refinement holds for S. This property is formalised in Theorem 6. So, if a product line refines another in terms of S, this is also true for any subset of S. We try to make S as larger as possible to potentiate our support, but this does not prevent one from choosing a smaller subset.

Theorem 6 (Partial refinement holds for subset)
For arbitrary product lines L and L', and sets of configurations S and S', if $L \sqsubseteq_S L'$ and $S' \subseteq S$, then $L \sqsubseteq_{S'} L'$.

Proof. For arbitrary product lines $L = (F, A, K)$ and $L' = (F', A', K')$, and sets of configurations S and S', by assuming $L \sqsubseteq_S L'$ and $S' \subseteq S$, which amounts to

$$\forall c \in S \cdot [\![K]\!]_c^A \sqsubseteq [\![K']\!]_c^{A'} \tag{3.7}$$

and

$$\forall c \in S' \cdot c \in S \tag{3.8}$$

We then have to prove

$$\forall c \in S' \cdot [\![K]\!]_c^A \sqsubseteq [\![K']\!]_c^{A'} \tag{3.9}$$

For an arbitrary c in S', we have to prove that $[\![K]\!]_c^A \sqsubseteq [\![K']\!]_c^{A'}$. Properly instantiating c in Predicate 3.8, we have that $c \in S$. So, we can instantiate c in Predicate 3.7 and this concludes our proof. □

It is also the case that, for an empty S, partial refinement trivially holds. However, this means that we would give no support, because we would guarantee that no products are refined. Developers might face this situation after applying consecutive transformations of partial refinement, if the intersection of the products refined in each step is empty (see Theorem 5). We could also try to provide some guarantee for products that are not in the scope of S, that constitutes the set of products possibly not refined. However, this would involve having a partial product refinement notion, and this is not in the scope of this work. So, for now we only give support for products in the scope of S.

3.2.2 Compositionality

To simplify reasoning about partial refinement, it is important to derive compositionality properties from our definition. These are useful, for example, when product lines main elements evolve separately to be later integrated to generate products. In this context, one might need to change a specific artifact, for instance, the FM, without changing the AM and CK. In this case, instead of using the definition to verify partial refinement after changes being applied to a specific artifact, we could use a theorem and simply verify that partial refinement holds. Developers could also modify different product line elements. We analyze these scenarios and whether such modifications preserve product line partial refinement. Compositionality theorems are provided in the existing refinement theory [BTG12], so it would be important to provide the same kind of modular support for partial refinement too. So, instead of using the definition, one could use the compositionality theorems provided here.

3.2.2.1 FM Partial Equivalence

We first analyse the FM. Developers often desire to change feature types and/or dependencies among them. For example, an optional feature may become mandatory. The refinement theory already provides support for FM refinement scenarios. According to the FM refinement notion (Definition 2), a FM refines another when the initial configurations are still present in the evolved FM. This holds for a number of scenarios, but changing a feature type from optional to mandatory is not considered a FM refinement. As shown in Theorem 1, FM refinement implies product line refinement. It is also true that FM refinement implies product line partial refinement (considering S to be all initial configurations), as we would be able to generate existing products in the new product line. However, we would like also to address scenarios such as a change in a feature type from optional to mandatory, that is not a FM refinement. In this case, the FMs may share configurations, but configurations not containing the affected feature possibly do not exist in the new product line. Thus, to provide more guarantees, we establish a partial FM equivalence notion, which allows the initial FM to have configurations not present in the final FM, differently from the FM equivalence and refinement notions, that require the semantics of the initial FM to be equal or a subset of the semantics of the final FM [BTG12]. According to Definition 9, the FMs only have a set of configurations S in common.

Definition 9 (Feature model partial equivalence)
For arbitrary feature models F and F', and a set of configurations S, F is equivalent to F' in terms of S, denoted by $F \cong_S F'$, whenever

$$\forall c \in S \cdot c \in [\![F]\!] \wedge c \in [\![F']\!].$$

Now, we would be able to support developers when transforming a feature type from optional to mandatory. Partial equivalence holds, if S is the set of configurations that already had the changed feature or do not have its parent. We should notice that FM equivalence and refinement lead to FM partial equivalence, but the opposite does not hold. As captured in Theorem 7, FM partial equivalence preserves product line partial refinement. Given a product line L, one can modify only the FM, by possibly adding, removing or modifying features and dependencies among them, but preserving a set of configurations S. Whenever only the FM is changed, there is still a partial product line refinement with respect to the same S. Since a product line by definition is well-formed [BTG12], we know that L is well-formed. However, we have no guarantee about L', more precisely, whether configurations that are in F' but are not in S lead to valid products. In exceptional cases, these could refer to features that are not in F' or do not obey rules, such as having a feature without having its parent. This is the reason for requiring well-formedness. Refinement holds because we are not checking products whose configurations are not in

S. Moreover, the weaker FM equivalence guarantees that S is in both FMs. Neither the AM nor the CK change. Therefore, we actually have exactly the same products if only checking configurations from S.

Theorem 7 (Feature partial equivalence compositionality)
For a product line $L = (F, A, K)$, a feature model F', and a set of configurations S, let $L' = (F', A, K)$. If $F \cong_S F'$ and L' is well-formed, then $L \sqsubseteq_S L'$.

Proof. For an arbitrary product line $L = (F, A, K)$, a FM F' and a set of configurations S, assume that $F \cong_S F'$. By Definition 9, this amounts to:

$$\forall c \in S \cdot c \in [\![F]\!] \wedge c \in [\![F']\!] \tag{3.10}$$

By Definition 8 we then need to prove that

$$S \subseteq [\![F]\!] \wedge S \subseteq [\![F']\!] \tag{3.11}$$

and

$$\forall c \in S \cdot [\![K]\!]^A_c \sqsubseteq [\![K]\!]^A_c \tag{3.12}$$

and

$$wfPL(L') \tag{3.13}$$

We can prove Predicate 3.11 directly from Predicate 3.10 and Predicate 3.13 is assumed in the theorem. Finally, Predicate 3.12 is trivially true from asset set refinement reflexivitiy (see Axiom 1). □

3.2.2.2 AM Partial Refinement

Similarly to the FM, the AM may also be modified separately. Previous work shows that the source code is modified in a higher frequency when compared to the FM or CK [DVDP14]. Some of these changes may be unsafe, like bug fix scenarios. In these cases, one not necessarily modifies the FM and the CK. The existing AM refinement compositionality (see Theorem 8) theorem would not be helpful in this case because it requires all assets from the initial AM to be refined by the evolved ones, that is not true in bug fix scenarios. For this reason, we also define partial refinement for the AM. As stated in Definition 10, an AM partially refines another when, for a finite set of names, refinement holds. More specifically, the AM resultant from filtering the original AM A according to a set of asset names ns (that is formalised as $A \triangleleft ns$, which expands to $\{(n : Name, a : Asset) | (n, a) \in A \wedge n \in ns\}$.) needs to be entirely refined (see Definition 3) by the AM obtained by filtering the new AM A' according to ns. In the case of bug fix scenarios, the new AM would refine the original one according to the assets not affected by the change.

Definition 10 (AM Partial Refinement)
For arbitrary asset mappings A and A', and a set of asset names ns, A' partially refines A in terms of ns, denoted by $A \sqsubseteq_{ns} A'$, whenever

$$(A \triangleleft ns) \sqsubseteq (A' \triangleleft ns),$$

AM partial refinement implies product line partial refinement, since products not containing any asset name in ns are not affected. As a product is represented as a set of assets only, we need to discover which products have assets whose names are in ns to precisely express the set of products refined after a change in an AM. Thus, in this context, we assume an evaluation function $(\!|\_|\!)$. This function could be seen as part of the CK semantics function, but instead of returning a set of assets, it returns assets and their names in the form of an AM (the submapping of the original AM containing only the assets used to build the product). This way, we are able to maintain the mapping and check if a product has an asset associated with a specific name in the AM.

Assumption 5 (Configuration knowledge evaluation)

$$(\!|\_|\!) : CK \to AM \to Configuration \to AM$$

The CK semantics function is then defined using the just defined evaluation function, as seen in Definition 11. The result of the semantics function is just collecting the assets in the resulting AM from the evaluation function, and ignoring their names. We use $A\langle\_\rangle$ to denote the relational image of an AM A. We are assuming that the evaluation function is responsible for the other steps in the product generation process.

Definition 11 (Configuration Knowledge Semantics)
Let $L = (F, A, K)$ be a product line. Then $[\![K : CK]\!] : \mathcal{F}[Asset] = (\!|K|\!)^A_c < dom((\!|K|\!)^A_c) >$

We do not desire an arbitrary evaluation function. It needs to obey some rules to avoid abnormalities. For this reason, we created axioms to guarantee that it is defined as we expect. They were derived as part of the compositionality proof. So, if they were not valid, compositionality theorems would not hold, and moreover, we could have abnormalities such as changing an asset name during the product generation process. Axiom 3 states that if the asset names of a determined product, which is the result of applying the CK semantics function, belong to a set of names ns, the result of the CK evaluation should be the same using the entire AM or the filtered AM according to ns. The reasoning is that if the other names are not present in the respective product, they can be discarded.

Axiom 3 (Unused assets do not influence CK semantics)
For an arbitrary AM A, a CK K, a finite set of asset names ns and a configuration c, if $dom((\!|K|\!)^A_c) \subseteq ns$, then $[\![K]\!]^A_c = [\![K]\!]^{A \triangleleft ns}_c$

It is also essential to guarantee that the resulting asset names from the generated products belong to the original asset mapping of the product line. Otherwise, we could have assets from other product lines, which does not make sense. For this reason, we formalize this in Axiom 4.

Axiom 4 (CK evaluation must preserve AM domain)
For arbitrary AM A, CK K, configuration c, $dom((\!|K|\!)^A_c) \subseteq dom(A)$

The third constraint is established in Axiom 5. It is related to Axiom 4. The evaluation function not only is forbidden to create new asset names, but for asset mappings with equal domain, the resulting domain after applying the evaluation function also needs to be equal. So, the evaluation function preserves equality over AM domain. We need both axioms because one does not exclude the other and we do not concretely define an evaluation function in this theory, since its definition depends on the CK structure, which is undefined at this theory level because we are assuming the CK as defined in Section 2.2.3. So, the CK to is an uninterpreted type, and its evaluation is generic here.

Axiom 5 (Evaluation preserves equality over AM domain)
For arbitrary AMs A and A', CK K, and configuration c, if $dom(A) = dom(A')$, then $dom((\!|K|\!)^A_c) = dom((\!|K|\!)^{A'}_c)$.

The three axioms just introduced are essential to avoid an arbitrary evaluation function, and they do not restrict the applicability of our theory, that is compatible with every notion that obeys these constraints. To guarantee that both compositional and transformational CKs satisfy these axioms, we instantiate this theory using both CK notions and prove them. The axioms are also important for stating and proving the AM compositionality theorem. We also found that all product line evolution scenarios analysed, which are detailed in Chapter 5 obey the three axioms. This confirms our intuition that they are reasonable to be assumed.

According to Theorem 8, partial AM refinement implies partial product line refinement. We calculate the set of configurations S for these situations based on the AM commonalities and differences. Configurations from S must not generate products containing the names that are not in ns, since as already discussed earlier in this section, these products are not refined. So, partial refinement does not hold for them. Alternatively, configurations that lead to products containing assets in the scope of ns are refined. To define S, we use a restriction operator $\restriction$ that takes the three elements of a product line (F, A, K) and a finite set of asset names ns. It then returns configurations whose products have only assets that are in ns, which is given by the set $\{c : Conf \mid c \in [\![F]\!] \wedge dom((\!|K|\!)^A_c) \subseteq ns\}$. However, it is not enough to filter configurations considering the original AM, since A and A' may be different. So, we need to define S as the intersection of filtering both AMs

according to ns (which is given by $((F, A, K) \restriction ns) \cap ((F, A', K) \restriction ns)$), unless we knew that A and A' have the same domain, but we do not assume this condition.

Theorem 8 (Asset mapping partial refinement compositionality)
For product lines $L = (F, A, K)$ and $L' = (F, A'K)$, and a finite set of asset names ns, if $A \sqsubseteq_{ns} A'$ then $L \sqsubseteq_S L'$, where $S = ((F, A, K) \restriction ns) \cap ((F, A', K) \restriction ns)$.

Proof. For an arbitrary PL $L = (F, A, K)$, an AM A' and a finite set of asset names ns. We have to prove that $L \sqsubseteq_S L'$, where $L' = (F, A', K)$ and $S = ((F, A, K) \restriction ns) \cap ((F, A', K) \restriction ns)$. According to Definition 8, $L \sqsubseteq_S L'$ expands to

$$S \subseteq [\![F]\!] \wedge \forall c \in S \cdot [\![K]\!]_c^A \sqsubseteq [\![K]\!]_c^{A'} \tag{3.14}$$

It is trivially true that $S \subseteq [\![F]\!]$, since this is expressed in the definition of S. So, then we need to prove that, for an arbitrary c in S, $[\![K]\!]_c^A \sqsubseteq [\![K]\!]_c^{A'}$. By properly instantiating K, A, ns, and c in Axiom 3, we have that $[\![K]\!]_c^A = [\![K]\!]_c^{A \lhd ns}$. The condition $dom((\!|K|\!)_c^A) \subseteq ns$ is satisfied due to S definition. Using Axiom 3 again, properly instantiated with K, A', ns, and c, we also have $[\![K]\!]_c^{A'} = [\![K]\!]_c^{A' \lhd ns}$. By replacing this in Predicate 3.14, we then need to prove that $[\![K]\!]_c^{A \lhd ns} \sqsubseteq [\![K]\!]_c^{A' \lhd ns}$. Using Definition 10, we have that $(A \lhd ns) \sqsubseteq (A' \lhd ns)$.

From Axiom 2, we have that $\forall am1, am2 \cdot am1 \sqsubseteq am2 \Rightarrow \forall K, c \cdot wf([\![K]\!]_c^{am1}) \Rightarrow wf([\![K]\!]_c^{am2}) \wedge [\![K]\!]_c^{am1} \sqsubseteq [\![K]\!]_c^{am2}$. Instantiating this equation with $am1 = A \lhd ns$ and $am2 = A' \lhd ns$, the first condition holds because (F, A, K) is a product line, and by definition, every product line is well-formed. So, this is enough to prove that $[\![K]\!]_c^{A \lhd ns} \sqsubseteq [\![K]\!]_c^{A' \lhd ns}$. □

Although we do not use Axiom 4 and Axiom 5 in this proof, they are used in a similar compositionality theorem in our online appendix[4] and also in the CHANGE ASSET template, which will be discussed in Chapter 4.

3.2.2.3 CK Partial Equivalence

Now, we analyse scenarios where the CK structure is changed in isolation and how this impacts the entire product line. Developers may need to modify the CK only and it is important to support them in these situations not only with our definition of partial refinement, but also with a CK partial equivalence notion. An example of such scenario would be adding a mapping between existing features and artifacts, assuming a compositional CK (see Section 3.2.2.3). In this case, the FM and the AM do not suffer any change. Only the CK is modified. There are other several possible modifications: deleting mappings, or even changing existing mappings in only one of the sides, either a feature expression or the respective assets.

[4] All PVS files and proofs are available at http://github.com/spgroup/theory-pl-refinement/tree/dev.

To address this and other evolution scenarios, we formalise a partial equivalence notion to represent partial refinement changes regarding the CK only. Notions to deal with refinement scenarios have already been proposed [BTG12]. However, most of the possible changes involving the CK are not refinements. Definition 12 generalizes all possible changes in the CK. We state that for a set of configurations S the respective products generated with the original and final CKs are equal. If the CKs are equivalent, S could be equal to the semantics of the FM, that is, the set of all valid configurations. In contrast, if the CKs are completely different, S would be empty, which is not desired becaused this would give no support.

Definition 12 (Configuration knowledge partial equivalence)
For arbitrary CKs K and K', and a set of configurations S, K is equivalent to K' in terms of S, denoted by $K \cong_S K'$, whenever

$$\forall am, c \in S \cdot [\![K]\!]_c^{am} = [\![K']\!]_c^{am}$$

Configuration Knowledge partial equivalence implies product line partial refinement. This is shown in the compositionality theorem (Theorem 9). Basically, considering an arbitrary product line L, let us suppose that a change is made to the CK of L but preserving the set of configurations S. We then obtain L', and we can say that L' partially refines L according to S as long as S is a subset of the valid configurations generated from the FM of L. Just to give an example, considering that we obtain K' by removing a mapping from a hypothetical feature P to an asset a present in K. In this case, S would be the set of configurations present in F that do not have P. Products containing P might not be refined, since they will not have the asset a as before. So, configurations containing P can not be included in S. Since K is equivalent to K' according to S and S is a subset of F configurations, L' refines L according to the same S.

Theorem 9 (Configuration knowledge partial equivalence compositionality)
For a product line $L = (F, A, K)$, a CK K', and a set of configurations S, let $L' = (F, A, K')$. If $K \cong_S K'$, $S \subseteq [\![F]\!]$ and L' is well-formed, then $L \sqsubseteq_S L'$.

Proof. For an arbitrary product line $L = (F, A, K)$, a CK K' and a set of configurations S, assume that $K \cong_S K'$ and $S \subseteq [\![F]\!]$. By Definition 5, this amounts to:

$$\forall am, c \in S \cdot [\![K]\!]_c^{am} = [\![K']\!]_c^{am} \tag{3.15}$$

By Definition 8 we then need to prove that

$$S \subseteq [\![F]\!] \tag{3.16}$$

and

$$\forall c \in S \cdot [\![K]\!]_c^{A} \sqsubseteq [\![K']\!]_c^{A} \tag{3.17}$$

We are already assuming Predicate 3.11. So, for an arbitrary c in S, we need to prove$[\![K]\!]_c^A \sqsubseteq [\![K']\!]_c^A$. By properly instantiating am and c in Predicate 3.15 with A and c, we have $[\![K]\!]_c^A = [\![K']\!]_c^A$. So, we can replace $[\![K']\!]_c^A$ by $[\![K]\!]_c^A$ and the proof follows from asset set refinement reflexivitiy (see Axiom 1). □

3.2.3 Combining different refinement and partial refinement notions

We also reason about compositionality in terms of combining different refinement notions, since the theories are not mutually exclusive; they are complementary. Thus, practitioners may desire to interleave refinement and partial refinement operations. For improvements or adding new features with behavior preservation, one can use the refinement theory. After that, developers may need to remove another feature, such as the feature removal scenario illustrated in Section 3.1. For these cases, the partial refinement notion should be used. Hence, the theories might be used interchangeably and we need to provide support in the sense that, when applying consecutive transformations, refinement still holds for a subset of products.

Refinement and partial refinement

When a partial refinement over S is followed by a refinement, we would ideally have partial refinement for products in S by transitivity. This is not possible because, in the refinement transformation, feature names do not matter, differently from the partial refinement notion. In fact, as Definition 7 admits configurations to change, even when S is equal to the set of all valid configurations, refinement is not necessarily a particular case of partial refinement.

To support interleaving of safe and unsafe changes, Definition 13 describes an alternative partial refinement notion that allows configurations to change according to a renaming function f. Then, given an initial configuration c, refinement holds for the product generated from $f(c)$. In a feature renaming situation, supposing that we change the feature name from P to P', f would be defined as $f(c) = c[P'/P]$. This function takes a configuration c and returns c if c does not have the feature P. Otherwise, it gives a new configuration c' as result, that is equal to c, except that every occurrence of P is replaced by P'.

Definition 13 (Weak partial refinement)
For arbitrary product lines $L = (F, A, K)$, $L' = (F', A', K')$ and a function $f : Conf \rightarrow Conf$, L' partially refines L in terms of f, denoted by $L \sqsubseteq_f L'$, whenever

$$\forall c \in dom(f) \cdot f(c) \in [\![F']\!] \wedge [\![K]\!]_c^A \sqsubseteq [\![K']\!]_{f(c)}^{A'}.$$

The partial refinement notion is a particular case of Definition 13 (when f is the identity function over S). Thus, this weak notion supports situations where configurations change, which are not covered by the default partial product line refinement notion (Definition 8). Since the weak definition is more general, we could have it instead of having both partial refinement relations. However, Definition 8 is less complex and developers only need to deal with the alternative one for feature renaming scenarios.

We have a function f as an index because allowing configurations to be arbitrarily modified having a set of configurations S as an index would lead to relations that are not transitive. Transitivity does not hold for such a definition because we have no control of the new configurations; they could be arbitrary. Thus, when applying consecutive refinements, we would not know if the refined configurations were the same as the ones refined in the first step. Hence, even assuming two refinement operations in terms of the same S, the transitivity does not hold for S. We desire to have a partial refinement relation that is a pre-order, since this considerably increases its applicability.

Similarly to Definition 7, this relation is also a pre-order, as we should support developers in the step by step refinement. In Theorems 10 and 11, we formalise the reflexivity and transitivity properties. A product line partially refines itself, according to Definition 13, when the function f is an identity. Otherwise, it makes no sense to compare different products in the same product line.

Theorem 10 (Weak partial refinement reflexivity)
For an arbitrary product line $L = (F, A, K)$, and a function $f : Conf \to Conf$, if f is the identity function and $dom(f) \subseteq [\![F]\!]$, then $L \sqsubseteq_f L$.

Proof. Let $L = (F, A, K)$ be an arbitrary product line. By Definition 13, we have to prove that, for an arbitrary c in $dom(f)$, $[\![K]\!]^A_c \sqsubseteq [\![K]\!]^A_{f(c)}$. Since f is the identity function, we can replace $f(c)$ follows from asset refinement reflexivity (see Axiom 1). □

For the transitivity property, the reasoning is similar to Theorem 5. Instead of giving refinement guarantees for the intersection of the two sets of configurations, we compose the two functions defined for each evolution step. So, suppose that we first refine a product configuration c_1 according to a function f resulting in c_2, and then another transformation is made to c_2 and we obtain c_3 by applying a function g. So, the product generated with c_3 refines the one obtained from c_1 in terms of the composite function gof.

Theorem 11 (Weak partial refinement transitivity)
For arbitrary product lines L_1, L_2, L_3, and functions $f : Conf \to Conf$ and $g : Conf \to Conf$, if $L_1 \sqsubseteq_f L_2$ and $L_2 \sqsubseteq_g L_3$, then $L_1 \sqsubseteq_{gof} L_3$.

Proof. Let $L_1 = (F_1, A_1, K_1)$, $L_2 = (F_2, A_2, K_2)$ and $L_3 = (F_3, A_3, K_3)$ be arbitrary product lines. Assume that $L_1 \sqsubseteq_f L_2 \wedge L_2 \sqsubseteq_g L_3$. By Definition 13, this amounts to:

$$\forall c \in dom(f) \cdot [\![K_1]\!]_c^{A_1} \sqsubseteq [\![K_2]\!]_{f(c)}^{A_2} \tag{3.18}$$

$$\forall c \in dom(g) \cdot [\![K_2]\!]_c^{A_2} \sqsubseteq [\![K_3]\!]_{g(c)}^{A_3} \tag{3.19}$$

We then have to prove that

$$\forall c \in dom(gof) \cdot [\![K_1]\!]_c^{A_1} \sqsubseteq [\![K_3]\!]_{g(f(c))}^{A_3} \tag{3.20}$$

For an arbitrary $c \in dom(gof)$, we need to prove that $[\![K_1]\!]_c^{A_1} \sqsubseteq [\![K_3]\!]_{g(f(c))}^{A_3}$. Since the domain of gof is equal to f domain, we can instantiate c in Predicate 3.18. We then have $[\![K_1]\!]_c^{A_1} \sqsubseteq [\![K_2]\!]_{f(c)}^{A_2}$. Properly instantiating c in Predicate 3.19 with $f(c)$, we then have $[\![K_2]\!]_f^{A_2}(c) \sqsubseteq [\![K_3]\!]_{g(f(c))}^{A_3}$. The proof then follows by asset set refinement transitivity (see Axiom 1). □

When one applies a partial refinement followed by a refinement, we have a weak partial refinement. A possible scenario of such situation is found, for instance, when one renames a feature, and then changes an asset in a non behavior-preserving way. Since not all products are refined because of the asset change operation, the domain of the function is only the set of configurations whose products do not have the changed asset. Suppose that feature P was renamed to P', L is the product line before these two operations and L' is the final product line, we then guarantee that $L \sqsubseteq_f L'$. The function f in this case would also be defined as $f(c) = c[P'/P]$, since in the asset change operation configurations were not changed. This notion is formalized in Theorem 12. When partial refinement is followed by refinement, there is a function that maps configurations from S to the final product line, so that weaker partial refinement holds.

Theorem 12 (Partial refinement and refinement)
For product lines L_1, L_2 and L_3 and a set of configurations S, let F_3 be the FM of L_3. If $L_1 \sqsubseteq_S L_2$ and $L_2 \sqsubseteq L_3$, then, for some function $f : S \to [\![F_3]\!]$, $L_1 \sqsubseteq_f L_3$.

Intuitively, this theorem is valid because there is always a function which maps configurations from the initial to the product line. When a refinement (Definition 7) occurs, we can derive a function that maps configurations. So, in this case we could say that there is a function $g : [\![F_2]\!] \to [\![F_3]\!]$ that maps configurations from L_2 to L_3. In the first case, when we have a partial refinement (Definition 8), we require that configurations do not change, differently from Definition 13. So, we can derive an identity function $I : S \to [\![F_2]\!]$, since the initial configuration is equal to the final one. Thus, $f : [S \to [\![F_3]\!]]$ in Theorem 12 would be the composition of g with the identity function I. Product refinement then holds by transitivity.

If the operations are conducted in the opposite order (refinement followed by partial refinement), the reasoning and end result are analogous, so we omit the details here.

Name aware refinement and partial refinement

Although the refinement and partial refinement correspondence is indirect, there is another definition for product line refinement that has basically the same meaning of Definition 7, but it does not allow changing the configurations. Consequently, this definition supports less scenarios when compared to Definition 7. Feature renaming, for instance, is not a refinement according to this notion. Since configurations are usually sets of feature names, when changing such names, configurations containing them are impacted.

Definition 14 (Name aware product line refinement)
For arbitrary product lines $L = (F, A, K)$ and $L' = (F', A', K')$, L' strictly refines L, denoted by $L \preceq L'$, whenever

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

Previous work has shown that this relation has similar properties to the refinement relation, like pre-order [BTG12]. Differently from product line refinement (Definition 7), this notion from Definition 14 is similar to the partial refinement notion (Definition 8). Since it does not allow any change in configurations, we can then establish a more direct relationship. For product lines L and L', name aware refinement implies partial refinement, provided that the set of configurations S is present in L. As a consequence, by transitivity, when a partial refinement is followed by a name aware refinement, this results in a partial refinement, as shown in Theorem 13. If the refinements are performed in the opposite order, the result is also a partial refinement.

Theorem 13 (Partial and stronger refinement)
For product lines L_1, L_2 and L_3 and set of configurations S, if $L_1 \sqsubseteq_S L_2$ and $L_2 \preceq L_3$, then $L_1 \sqsubseteq_S L_3$.

Commutativity of name aware refinement and partial refinement

Finally, we also reason whether the name aware refinement and partial refinement transformations lead to the same product line when applied in different orders, and we demonstrate that this property holds. For instance, given a product line L_1, suppose that a developer performs a name aware refinement, such as locally refactoring an asset, obtaining L_3 and then partially refines the product line by removing a feature, obtaining L_4. Figure 3.1 represents a commutative diagram that shows that if we instead first apply this same partial refinement operation (yielding L_2) and then refine the asset, we obtain the same L_4. Thus, in this case, the order in which the transformations are applied does not matter.

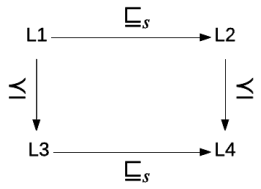

Figure 3.1 – Commutative diagram

Properties like this one reflect what happens during development, where practitioners might want to apply several different operations consecutively and it is helpful to be sure that applying refinements in a different order can produce the same result. We formally derive and prove two theorems from the commutative diagram structure shown in Figure 3.1. Although we do not present the proofs here, they are available in our online appendix.

In Theorem 14, we give support in case developers are doing first a partial refinement and then a name aware refinement. The theorem establishes that there is an alternative way to obtain the same resulting product line, by performing the corresponding operations in the opposite order. Theorem 15 is analogous. This theorem has an extra condition when compared to the first one. This property only holds if S is a subset of the valid configurations generated by the initial product line L_1. This condition is necessary, as otherwise we could have invalid products, since invalid configurations may not obey dependency rules among features. Thus, it does not make sense to refine a product line in terms of an S that is not part of the product line configurations.

Theorem 14 (Partial refinement and name aware refinement commute (1))
For product lines L_1, L_2 and L_4, and a set of configurations S, if $L_1 \sqsubseteq_S L_2$ and $L_2 \preceq L_4$, then, for some product line L_3, we have $L_1 \preceq L_3 \wedge L_3 \sqsubseteq_S L_4$.

Theorem 15 (Partial refinement and name aware refinement commute (2))
For product lines L_1, L_3, L_4 and a set of configurations S. Let F_1 be the FM of L_1. If $S \subseteq [\![F_1]\!]$, $L_1 \preceq L_3$ and $L_3 \sqsubseteq_S L_4$, then, for some product line L_2, we have $L_1 \sqsubseteq_S L_2 \wedge L_2 \preceq L_4$.

3.3 Discussion

In the previous section, we provide a correspondence between the partial refinement and stronger refinement relations, through a commutative diagram illustrated in Figure 3.1. We believe that the diagram also holds if we replace $\preceq$ by $\sqsubseteq$. Nevertheless, we do not formalize the partial refinement and refinement commutativity in this sense. To prove the correspondent theorems, we would possibly need to enrich our theory, where instead of having only relations connecting product lines, we would also have transformation operations expressing how product lines change. Instead of considering just sets of product

lines, our encoding would have also to consider sets of transformations or changes among product lines. The proof would then be made by induction over the set of possible transformations.

As mentioned in Section 3.2.3, the partial refinement relation is far less similar to the product line refinement notion than to the stronger one, because only the second one allows changes in configurations, whereas the others do not. Thus, the proof regarding the commutative property relating the refinement and partial definitions would become more complex and we would need to provide a valid L_3 assuming arbitrary L_1, L_2 and L_4 and that $L_1 \sqsubseteq_S L_2$ and $L_2 \sqsubseteq L_4$. Intuitively, L_3 is the result one obtains by applying the same operation performed in L_2 that resulted in L_4. To express this formally, however, we need a function that maps product lines before and after the evolution process, such that assuming a transformation function $t : [PL- > PL]$ and $L_4 = t(L_2)$, we can then instantiate L_3 with $t(L_1)$. Since we did not formalize this theory yet, we do not have the commutative diagram (Figure 3.1) using the refinement definition.

3.4 PVS Encoding

To understand how the presented theory is encoded in PVS and relates to the product line refinement theory, we discuss the impact and the growth of the product line refinement theory in PVS.[5] In Figure 3.2, it is possible to observe the dependencies among theories and their hierarchy. The new theories that were created due to the inclusion of partial refinement are highlighted in grey colour. As the partial refimement concept builds on the existing product line theory and concepts, all assumptions existing there, like asset set refinement preorder (see Section 2.2), are also required here. On top of that, we have assumptions specifically to partial refinement, like the axioms presented in Section 3.2.2.2. We prove all the new properties to guarantee that they are valid and consistent with the existing theory. In the remainder of this section, we explain each theory in more detail. Some of them deal with general notions of FM, AM and CK (*PartialRefBasics*, *PartialRefDefault*, *PartialRefWeaker*, *PartialRefinement* and *PartialAMCompositionality*), and ideally they would be valid with any product line definition that has the three elements. These five theories have six parameters: FM, Asset, Asset Name and CK types, and also FM and CK semantics functions. Thus, one can instantiate it with concrete languages for the three product line elements and implement semantics functions. One needs to provide concrete notions, and semantics functions for the FM and CK. The other two theories are specific for compositional and transformational CK respectively.

- **PartialRefBasics**: in this theory, partial refinement/weaker equivalence notions for

[5] All PVS files and proofs are available at http://github.com/spgroup/theory-pl-refinement/tree/dev.

the FM, AM and CK are defined. It imports the preexisting refinement theory that defines these basic types representing the three main elements of the product line.

- **PartialRefDefault**: this theory contains the main partial refinement definition (Definition 8). It uses **PartialRefBasics**, as we analyze whether the transformations applied to the FM and the CK in separate lead to product line partial refinement. Furthermore, we also reason about refinement and partial refinement transformations being applied consecutively. So, theorems like Theorems 14 and 15 are defined in this theory.

- **PartialRefWeaker**: this is analogous to **PartialRefDefault**, but here we are dealing with the weaker partial refinement notion (Definition 13).

- **PartialRefinement**: in this theory, we analyse how the definitions presented previously (default and weaker) relate to each other. Basically, here we establish that if the function f in the weaker defintion is the identity function, this definition is equivalent to Definition 8.

- **PartialAMCompositionality**: as we discussed in Section 3.2.2.2, we assume the existance of the evaluation function to reason about AM compositionality. For this reason, this is expressed in a separate theory, as we do not need the evaluation function for the other concepts. We do not allow the evaluation function to be arbitrarily defined. As discussed in Section 3.2.2.2, it must obey a set of constraints, like not generating assets not present in the AM of the product line being evaluated.

- **PartialAMRefInstComp**: this is an instantiation of **PartialAMCompositionality**, where we deal with the general CK notation introduced in Section 2.2.3. This theory is essential to certify that assumptions made regarding an arbitrary CK would be valid for the compositional CK. So, we prove that Axioms 3, 4 and 5 also hold for compositional CKs.

- **PartialAMRefInstTrans**: analogous to **PartialAMRefInstComp**, but deals with transformational CKs.

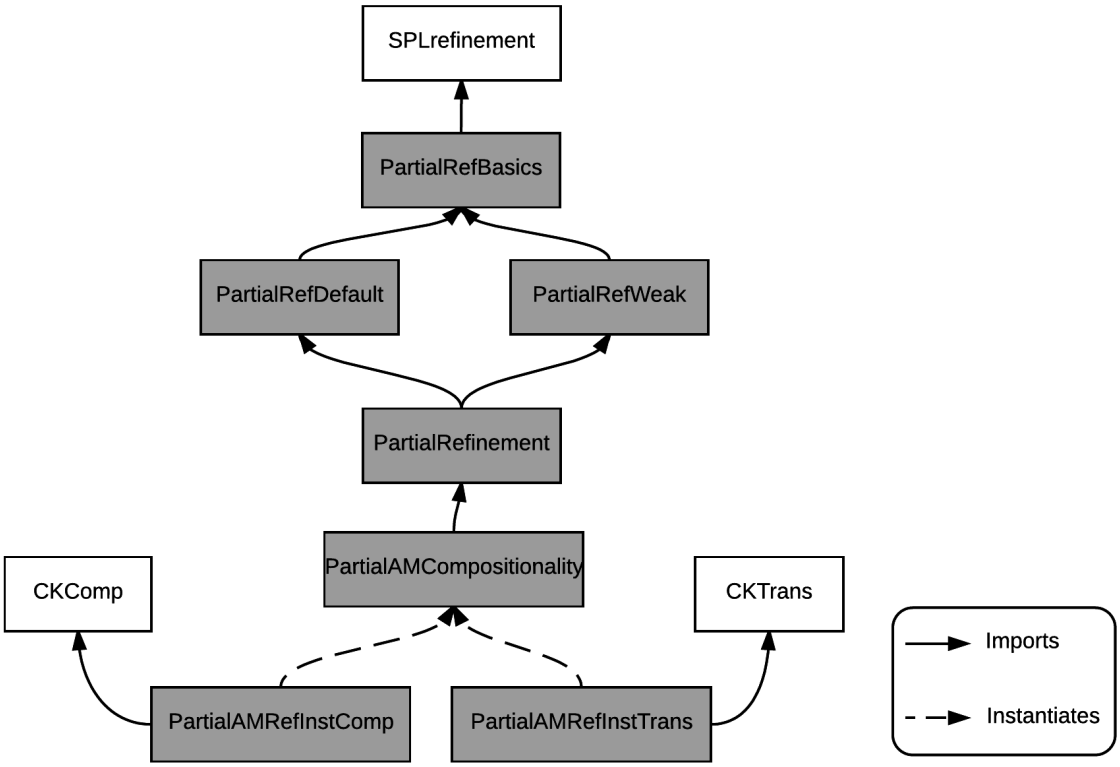

Figure 3.2 – PVS theories hierarchy

4 PARTIALLY SAFE EVOLUTION TEMPLATES

As mentioned in Chapter 3, the partial refinement theory can be applied to several contexts (possibly even more than the refinement theory). In this chapter, we exemplify such contexts and define templates that are abstractions of recurrent practical evolution scenarios. We defined such templates based on preexisting refinement templates [NBA+15, BTG12], by changing conditions to allow partially safe changes. Templates are helpful because they provide guidance on how to evolve a product line guaranteeing safe evolution for a subset of the products. Moreover, developers do not need to understand the theory for these scenarios; they just need to understand the templates. Templates also avoid errors during the evolution process and increase developers confidence.

A template has a left-hand side pattern (LHS) and a right-hand side pattern (RHS). They correspond to abstractions that capture properties of the initial and evolved product lines, respectively. We make use of meta-variables to represent the initial and evolved product line elements. In case one follows the syntactic and semantic rules established by templates, it is guaranteed that partial refinement holds for a specified subset of products S.

The developer does not choose the value of S; it is defined based on the FM, AM and CK of the product lines in the templates. Establishing S this way helps to understand the change impact, since products in the scope of S are not impacted. We should remember that, as we stated in Theorem 6, product line partial refinement holds for any subset of S, given that it holds for S. So, one may choose to work with a smaller subset. In Section 2.1.3, we discuss two notations for the CK structure: compositional and annotative. In this chapter, we provide a set of templates that are compatible with each of these notations (Section 4.1 and Section 4.2). Some templates are not included in these two groups (Section 4.3), since they do not depend on any particular CK notation. In this case, we assume a more general CK like the one introduced in Section 2.2.3.

4.1 Compositional CK

In this section, we present templates that use the compositional CK notation (see Section 2.1.3). In this notation, the CK is assumed to be a table-like structure where there are two columns. The left column has feature expressions and the right column has asset names. In the following, we analyze a number of possible scenarios of partially safe evolution.

Remove feature

We first analyze feature removal situations, which is an usual scenario in a product line development context. One often decides to exclude features for diverse reasons [PTD+15]; for instance, they are no longer used or not needed by customers. We define the REMOVE FEATURE template in Figure 4.1. Products that did not have the removed feature in the original product line keep the same behavior, and the others might not be refined. In this template, the three product line elements are changed. By syntactically analyzing the REMOVE FEATURE template in Figure 4.1, we observe that the initial FM (F), has the O feature, which is removed, and consequently, F', including cross-tree constraints, does not have it. We also notice that O is P's child. Nothing else is changed in the FM, which might have other features beyond O and P. We assume that the initial CK has references to O, so from the LHS to the RHS, every row in the CK (like the one containing e' and n') referencing O is removed. If the CK has no references to O, the feature could be removed directly but this scenario would be also a product line refinement. The AM also loses a set of mappings, like a' which implements O.

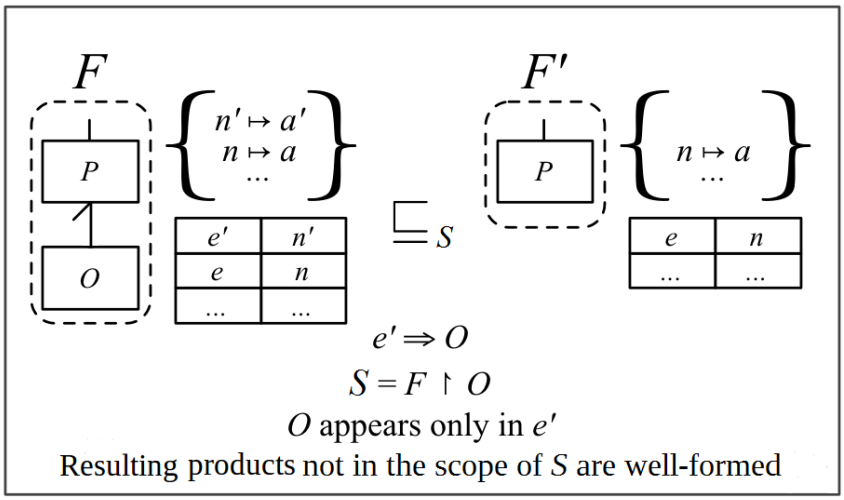

Figure 4.1 – REMOVE FEATURE compositional partial refinement template

The guarantees provided by the template only hold if some conditions are valid. We need to make sure that when e' is true, O has been selected, otherwise it would make no sense to exclude it from the CK. To do so, we require that $e' \Rightarrow O$. Consequently, a' is removed, since it must be in a product whenever O is selected. We could use simply O instead of e', but this would restrict the template applicability. When using e', we are allowing any expression where O is true. When a feature is removed, the intuition is that the products that did not have the respective feature do not change, so behavior is preserved. The other ones might not be compatible to any product in the new product line because they lose functionality, unless a' adds no extra behavior to a product. To specify S to capture that, we make use of the $\restriction$ operator, which filters configurations from a FM according to a feature expression. This expression may contain feature names and logical

operators, such as *and*, *not* and *or*. The expression $P \wedge Q$, for instance, is satisfied by a configuration c when c has the P and Q features. For an arbitrary FM F and a feature expression e, we use $F \upharpoonright e$ to denote the set of configurations in $[\![F]\!]$ that do not satisfy e. Thus, we specify S as $F \upharpoonright O$, giving refinement guarantees only for product configurations that are in F and do not include O. Since we only remove the line containing e' and n' from the CK, it is required that O does not appear in e and other CK lines, otherwise the feature would not be completely removed. Finally, we also need a well-formedness condition to guarantee that the products not refined (the ones that had O in the initial product line) remain well-formed. Since we assume that assets are removed from the initial to the evolved product line, we can not guarantee that existing products remain well-formed, except those refined.

The REMOVE FEATURE template does not assume that O is a leaf feature. However, when O is removed, the subtree under O is also removed according to the template. One might need to remove O, but keeping its children. So, we would need another template, which would be a variation of the template illustrated in Figure 4.1, to deal with such scenario and this is part of our future work.

Strictly, this template does not match the example discussed in Section 3.1, but it is compatible with a slight variation of the template, where two assets are removed. To illustrate that, we instantiate the meta-variables for the example. In this case, F is instantiated with the initial Linux VM containing *LEDS_RENESAS_TPU*, and F' is the resultant VM without this feature. The initial CK is instantiated with the Linux CK, including the line shown in Listing 3.2 and the changed CK is the same except for this mapping. The Linux AM could be represented by mappings between the file names to their respective contents. Using the feature removal example, n' would be *drivers/leds/leds-renesas-tpu.c* and *drivers/leds/leds-renesas-tpu.h*, and a', the respective contents of these source code files. The other mappings, such as $n \mapsto a$, correspond to other source file names and the respective contents. The new AM is obtained from the initial by removing the mapping $n' \mapsto a'$, which corresponds to the implementation of the removed feature. It is true that $e' \Rightarrow O$, since e' is *LEDS_RENESAS_TPU*. This feature appears only in e', since we did not find occurrences of this feature in the remaining items of the CK. Assuming that the resulting product line is well-formed, all conditions are satisfied. S is $F \upharpoonright$ *LEDS_RENESAS_TPU*. Thus, for these configurations the refinement holds. The other products are not refined since they have the removed feature, thus not preserving behavior. Differently from product line refinement (Definition 7), that requires every product in the initial product line to be compatible with at least one product in the new product line, partial refinement requires refinement for a subset of the initial products. So, in this case, only products without *LEDS_RENESAS_TPU* are refined.

Formalization

All of the templates presented in this chapter were encoded and proved in the PVS system. However, we do not present all specifications and proofs here, but all PVS files are available online.[1] In this section, to illustrate our formalization approach, we present the REMOVE FEATURE template formalisation. To specify partially safe templates, we follow the same strategy found in previous works [GMB05, BTG12, NBA+15, TABG15]. So, we first define the *syntax* and *conditions* predicates to encode the information present in the template. All syntactic similarities and differences regarding the initial and final product lines form the *syntax* predicate. In the REMOVE FEATURE template, the three product line elements are presented in detail, so we define syntactic rules for all of them. Preconditions like well-formedness rules are specified with the *conditions* predicate.

To specify the *syntax* predicate, we make use of pre-existing functions defined for concrete FM and CK. For example, the FM we are dealing with has a set of features and a set of formulae. So, for the REMOVE FEATURE template, we state that F' ($fm2$ in the formalization) formulae are all formulae from F ($fm1$ in the formalization), except those that have O. This is formalized as $formulae(fm2) = remove(O, formulae(fm1))$, and $remove(O, formulae(fm1))$ expands to $\{f : Formulae \mid f \in formulae(fm1) \wedge O \notin names(f))\}$. It would not make sense to allow these formulae to be part of F', since the O feature is removed. So, this guarantees that the O feature does not appear in any formula in F. Besides that, we also require that features from F' are exactly the ones from F, except for O. As Figure 4.1 shows, P and O need to be features from the initial FM. As a consequence of the second condition, P is also in F'.

We also describe the AMs and CKs. As we have just discussed, the removed feature does not need to be implemented by one asset only, nor be present in only one expression in the CK. We assume that several items in the CK and in the AM may be removed, and we represent these two sets with the *its* and *pairs* variables, respectively. Thus, the specification is actually more general than the template shown in Figure 4.1. Basically, the intitial AM must be an extension of the final one, with the AM *pairs*. The $\oplus$ operator can be simplified to $pairs \cup (am2 - dom(pairs))$, where $am2 - dom(pairs)$ is the set of pairs that belong to $am2$ whose names are not in $dom(pairs)$. The CK is represented as a set of items in the compositional language, so we say that K has every item from K' and also the removed items in *its*. Finally, we also need to certify that every configuration satisfies the $O \Rightarrow P$ expression, as this is derived from the template.

[1] http://github.com/spgroup/theory-pl-refinement/tree/dev

$$
\begin{aligned}
&syntax(fm1, fm2, am1, am2, ck1, ck2, P, O, its, pairs) : bool = \\
&\quad formulae(fm2) = remove(O, formulae(fm1)) \wedge \\
&\quad features(fm2) = remove(O, features(fm1)) \wedge \\
&\quad P \in features(fm1) \wedge \\
&\quad O \in features(fm1) \wedge \\
&\quad am1 = am2 \oplus pairs \wedge \\
&\quad ck1 = ck2 \cup its \wedge \\
&\quad \forall c \in [\![fm1]\!] \cdot sat(O \Rightarrow P, c)
\end{aligned}
$$

We also define preconditions. The first one is to define S as $F \restriction O$, which means the set of configurations that are in F semantics, but do not satisfy O. It is also necessary to make sure that every expression from its implies O, which is represented by the e' variable in Figure 4.1. Regarding the CK, it also required that O does not appear in other lines. So, we establish that configurations from the initial FM satisfy expressions from its if and only if they have the feature O. The second condition is regarding well-formedness. Developers must be sure that initial products containing O implementation remain well-formed. Finally, we have a condition regarding its and $pairs$. We require that the remaining features do not have their implementation removed, by guaranteeing that if an item does not belong to its, its respective assets, obtained by $getRS(item)$, are not in $pairs$, and consequently not removed.

$$
\begin{aligned}
&conditions(fm1, S, its, pairs, P, O, ck, ck2, am2) : bool = \\
&\quad S = F \restriction O \wedge \\
&\quad \forall c \in [\![fm1]\!] \cdot \\
&\qquad \forall exp \in exps(ck) \cdot \\
&\qquad\quad sat(exp, c) \Rightarrow exp \in exps(its) \Leftrightarrow sat(O, c) \wedge \\
&\qquad c \notin S \Rightarrow wf([\![ck2]\!]_c^{am2}) \wedge \\
&\quad \forall item \in ck \cdot item \notin its \Rightarrow \forall an \in getRS(item) \cdot an \notin dom(pairs)
\end{aligned}
$$

The template is already encoded, but we do not prove it directly. We define a strategy for an stepwise proof. First, we prove that configurations in S do not satisfy any expression in its. This guarantees that, when evaluating the CK, items associated with the O feature are not included, and consequently artifacts that implement O are not also present. This is specified in Lemma 1. The $evalCK$ function yields K items whose feature expressions are satisfied in the configuration c.

Lemma 1 (Items from its are not included)
For product lines $L = (F, A, K)$ and $L' = (F', A', K')$, a set of configurations S, a set of

items its, an AM $pairs$ and features P and O, if syntax(F,F',A,A',K,K',P,O,its,$pairs$) and conditions(F,its,$pairs$,P,O,K) hold, then

$$\forall c \in S \cdot \forall item \in evalCK(K, c) \cdot item \notin its$$

Proof. For arbitrary product lines $L = (F, A, K)$ and $L' = (F', A', K')$, set of configurations S, set of items its, AM $pairs$ and features P and O, we assume that $syntax(F, F', A, A', K, K', P, O, its, pairs)$ and $conditions(F, its, pairs, P, O, K)$ predicates hold. Expanding the conditions predicate, we have that:

$$S = \{c | c \in [\![F]\!] \wedge \neg sat(O, c)\} \tag{4.1}$$

and

$$\forall c \in [\![F]\!] \cdot \forall exp \in exps(K) \cdot sat(exp, c) \Rightarrow (exp \in exps(its) \Leftrightarrow sat(O, c)) \tag{4.2}$$

We then have to prove that, for an arbitrary c in S,

$$\forall item \in evalCK(K, c) \cdot item \notin its \tag{4.3}$$

For an arbitrary item it from $evalCK(K, c)$, we have to prove that $it \notin its$. Using set theory and properly instantiating c, we simplify Predicate 4.1 to $c \in [\![F]\!] \wedge \neg sat(O, c)$. Properly instantiating Predicate 4.2 with c and $exp(it)$, we have that $sat(exp(it), c) \Rightarrow (exp(it) \in exps(its) \Leftrightarrow sat(O, c))$. The expression $sat(exp(it), c)$ holds, because it belongs to $evalCK(K, c)$, that expands to $it \in K \wedge sat(exp(it), c)$. So, we can conclude $exp(it) \in exps(its) \Leftrightarrow sat(O, c)$. As we had $\neg sat(O, c)$, we conclude that $\neg exp(it) \in exps(its)$. So, we prove that $it \notin its$ and conclude our proof. □

We also introduce Lemma 2, to prove that, for products in S, assets resulting from the evaluation of the initial CK K ($eval(K, c)$ yields all asset names mapped to feature expressions that are satisfied according to the configuration c) do not belong to the removed assets from $pairs$. This means that assets implementing the removed feature are not present in the evolved CK. This lemma is related to Lemma 1, where we show that the evolved CK does not have any expression involving the removed feature. Although we do not present in detail here, all definitions used in this proof can be found in our Git repository.[2]

Lemma 2 (Assets from $pairs$ are not included)
For product lines $L = (F, A, K)$ and $L' = (F', A', K')$, a set of configurations S, a set of items its, an AM $pairs$ and features P and O, if syntax(F,F',A,A',K,K',P,O,its,$pairs$) and conditions(F,its,$pairs$,P,O,K) hold, then

$$\forall c \in S \cdot \forall an \in eval(K, c) \cdot an \notin dom(pairs)$$

[2] http://github.com/spgroup/theory-pl-refinement/tree/dev

Proof. For an arbitrary configuration c in S and asset name an in $eval(K, c)$, we have to prove that $an \notin dom(pairs)$. If an is in $eval(K, c)$, by expanding $eval$, we have that, for some item $it \in K$, $an \in getRS(item) \wedge item \in evalCK(K, c)$. Properly instantiating Lemma 1, we have that $it \notin its$. Expanding the *conditions* predicate leads us to

$$\forall item \notin its \cdot \forall an \in getRS(item) \cdot an \notin dom(pairs) \tag{4.4}$$

Properly instantiating it and an in Predicate 4.4, we have that $an \notin dom(pairs)$ and conclude our proof. □

We are now able to prove that removing a feature, given the *syntax* and *conditions* predicates previously established, lead to product line partial refinement. This idea is formalised in Theorem 16. Essentially, when a feature is entirely removed from a product line, and no elements regarding the remaining features are removed, we say that the evolved product line partially refines the initial one for configurations that do not have the removed feature.

Theorem 16 (Removing a feature is a partial refinement)
For product lines $L = (F, A, K)$ and $L' = (F', A', K')$, a set of configurations S, a set of items its, an AM $pairs$ and features P and O, if syntax(F,F',A,A',K,K',P,O,its,$pairs$) and conditions(F,its,$pairs$,P,O,K) hold and $\forall c \notin S \cdot wf([\![K']\!]_c^{A'})$, then $L \sqsubseteq_S L'$, where $L' = (F', A', K')$.

Proof. We have to prove that $L \sqsubseteq_S L'$, which, according to Definition 8, expands to

$$S \subseteq [\![F]\!] \wedge S \subseteq [\![F']\!] \tag{4.5}$$

and

$$\forall c \in S \cdot [\![K]\!]_c^A \sqsubseteq [\![K']\!]_c^{A'} \tag{4.6}$$

To prove Predicate 4.5, we first expand the restriction operator $\upharpoonright$ in our assumption about S, which leads us to $\forall c \in S \cdot c \in [\![F]\!] \wedge \neg sat(O, c)$. So, we can conclude that $S \subseteq [\![F]\!]$. To prove that $S \subseteq [\![F']\!]$, we expand the FM semantics function $[\![\_]\!]$, and we have to prove that all features and formulae in F' belong to F. $[\![F']\!]$ expands to $\{c : Configuration \mid satImpConsts(F', c) \wedge satExpConsts(F', c)\}$. Expanding these two predicates, we have to prove that:

$$\forall c \in S \cdot \forall (n : Name) \in c \cdot n \in features(F') \tag{4.7}$$

and

$$\forall c \in S \cdot \forall (f : Formula) \in formulae(F') \cdot sat(f, c) \tag{4.8}$$

From S definition, we have that $S \subseteq [\![F]\!]$, which amounts to $\forall c \in S \cdot \forall (n : Name) \in c \cdot n \in features(F)$. So, to prove Predicate 4.7, we need to prove that n can not be O. Otherwise,

since it belongs to $features(F)$, it will also be in $features(F')$. Also from S definition, we conclude that c does not satisfy O. So, O can not be in c's names. Predicate 4.8 also holds because it holds for $formulae(F)$ and $formulae(F') \subseteq formulae(F)$.

We then need to prove Predicate 4.6. Assuming an arbitrary $c \in S$ and expanding the CK semantics function, this simplifies to $A{<}eval(K, c){>} \sqsubseteq A'{<}eval(K', c){>}$. The difference between K and K' is the set of items (its) that belong to K but not to K'. Expanding $eval(K, c)$ results in $getRS(evalCK(K, c))$. By using Lemma 1, we conclude that all CK items resulting from $evalCK(K, c)$ do not belong to its. As the other items, except its, are equal, K and K' evaluation results in the same items for configurations in S. So, we have that $getRS(evalCK(K, c)) = getRS(evalCK(K', c))$.

The difference between A and A' refers to $pairs$. In Lemma 2, we show that there are no assets from $pairs$ resulting from the CK K evaluation. So, the assets obtained by the image in A and A' are the same and do not refer to $pairs$, so it makes no difference in obtaining the image in A and A', which is formally expressed as $A{<}eval(K, c){>} = A'{<}eval(K, c){>}$. Given that $[\![K]\!]_c^A = [\![K']\!]_c^{A'}$ for an arbitrary c in S, the proof folows by asset set reflexivity (see Axiom 1).

As every product line is well-formed by definition, PVS generates two TCCs (type check conditions) regarding the well-formedness of the evolved FM and product line. We are able to prove that the final FM is well-formed because, since the initial FM is well-formed, and the only transformation is removing a feature, F' is also well-formed. Regarding the final product line well-formedness, we need to prove that all products are well-formed, according to Definition 6. There are two scenarios to be considered: if a product is in the scope of S, we guarantee its well-formedness because it is equal to an initial product, as we have just proved in this theorem. For products that do not belong to S, we make use of the condition requiring that all products in L' which are not in S should be well-formed. So, we are able to prove that L' is well-formed. □

Adding and Removing assets

Another possibility for partially evolving a product line is adding new assets and their references in the CK. This might not be a completely partially safe evolution scenario, since some features will be associated to new assets that likely will change their behavior. Thus, we characterize it as a partially safe evolution scenario and give refinement guarantees only for products that are not extended with the new assets. The template illustrated in Figure 4.2 captures this scenario. It requires the FM not to change. The AM in PL' is an extension of the previous AM with new mappings, and new items are added to the CK. Note that nothing is removed. We are only assuming additions to both entities.

The set of refined products in this case corresponds to the ones that do not have the

new assets, so they are not affected. We express the respective configurations by filtering the ones that do not satisfy expressions in its, which is the set of items that might refer to the added assets. So, the set of configurations that belong to F semantics and satisfy the expressions from its corresponds precisely to the products that were extended. Another condition is that the assets in m must not appear in K. This is necessary to guarantee that the added assets are not referred by existing expressions in K. We also require that products from PL' that are not in S must be well-formed. Since the AM and CK are modified, we have no control of such additions, so we demand well-formedness.

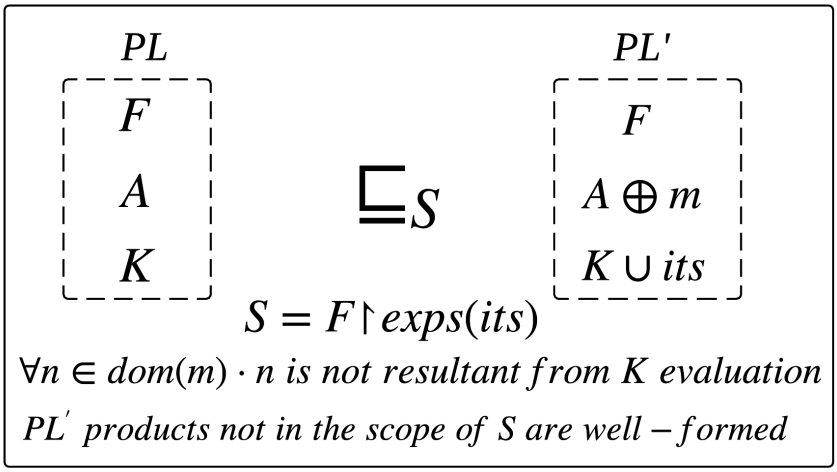

Figure 4.2 – ADD ASSETS partial refinement compositional template

We do not require that A and K are modified, since m and its could be empty. However, in case unused assets are added (by changing only the AM), $exps(its)$ would return the emptyset and, consequently, S would be equal to F semantics since all products would be refined. So, this would be the same as using the ADD UNUSED ASSETS safe evolution template [NBA+15]. Moreover, if the three elements of the product line are modified and the added assets are only associated to the new feature, this would also be a safe feature addition scenario (ADD NEW OPTIONAL FEATUREtemplate) [NBA+15], provided that the added feature is optional. However, in cases of unsafe feature additions, such as mandatory feature additions, one could obtain support of the partial refinement theory by dividing this modification in two parts: first, only the FM would be modified by adding a mandatory feature using the ADD NEW MANDATORY FEATURE template [NBA+15], and then adding the respective assets by using the template illustrated in Figure 4.2. The first step is safe, since it is a FM refinement scenario. The second step, however, is partially safe because the mandatory feature added in the previous step had no implementation, and in the resulting product line part of the products are affected by these new assets. The guarantee would hold for products that do not have the added feature, as asserted by Theorem 13, that guarantees refinement for a set of products after performing a partial refinement followed by a feature name aware refinement.

This template could be applied not only to asset additions scenarios, but also asset removals. If we read the template in reverse order (from right to left), we have the REMOVE ASSETS template. In this case, elements are possibly removed from the AM and the CK. We also guarantee that at least product configurations not satisfying expressions in its are refined. So, S should be as in the template. As in the asset additions scenarios, we also need the two other conditions to certify that assets from m are completely removed, so they do not appear in K and that the transformation preserves well-formedness for the products potentially not refined. We do not show this template here since it is analogous to the ADD ASSETS template, but it can be found in our online appendix.[3]

Changing, Adding and Removing CK lines

Finally, we consider a template in which only the CK changes, which is the CHANGE CK LINE template, shown in Figure 4.3. As explained in Section 3.1, the CK maps feature expressions to asset names. In a product line development context, for instance, one might need to replace a feature that is associated with an asset name by another one. The opposite may also happen, which would be to change the respective asset name, or even changing both. To cover these situations, we can use this template. We assume an expression e to be changed to e' and a set of asset names ns being changed to ns'. The other items found in the initial CK (its) are also in the new CK.

We assume that the new expression e' only contains feature names that are in F, otherwise we would not have a well-formed expression. Since we change e to e', products whose configurations satisfy e or e' are not refined. Suppose that e is equal to a hypothetical feature $F1$ and a user decides to transfer the implementation of $F1$ to a feature named $F2$. To apply such change, one has to change the expression from $F1$ to $F2$ and this means that the assets that were implementing the former now implement the latter. Consequently, both features are affected. So, S should have configurations that do not satisfy both, which means configurations that do not have both $F1$ and $F2$. Therefore, we define S as the intersection of the configurations that do not satisfy e and e'. As they are not affected, we can guarantee that refinement holds for the products that do not enable e and e'. We also assume that products from PL' not in S are well-formed because we can not prove such condition assuming arbitrary e' and ns'.

We also consider variations of the CHANGE CK LINE template. In Figure 4.4, we have the ADD CK LINES template. The CK is the only element that may be modified, and the set of items its are added to the product line. The effect is that product configurations satisfying expressions from its are potentially not refined. For this reason, we define S as the set of configurations that belong to F semantics but do not satisfy expressions from its. The products in the scope of S remain the same. In the same way as we do with the

[3] http://github.com/spgroup/theory-pl-refinement/tree/dev

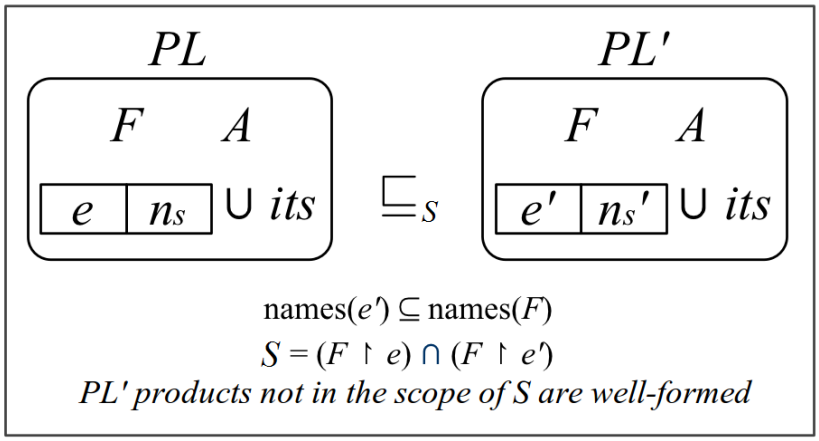

Figure 4.3 – CHANGE CK LINE partial refinement compositional template

ADD ASSETS template, if we read the template from Figure 4.4 from right to left (and also assuming well-formedness for PL instead of PL'), we have the REMOVE CK LINES template.

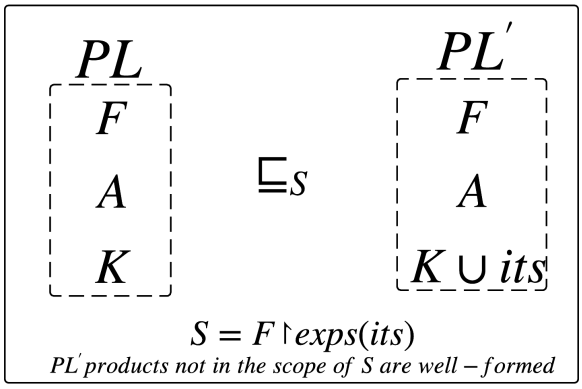

Figure 4.4 – ADD CK LINES partial refinement compositional template

We have just presented six compositional partial refinement templates: REMOVE FEATURE, ADD ASSETS, REMOVE ASSETS, CHANGE CK LINE, ADD CK LINES and REMOVE CK LINES. We discussed the REMOVE FEATURE formalization (see Section 4.1), and the others can be found in our online appendix.[4] In the following, we show templates compatible with transformational CKs and templates that are compatible with both CK notations.

4.2 Transformational CK

As shown in the previous section, there are several possible partially safe evolution scenarios that developers might face in practice. Here, we deal with the same scenarios, but assuming that the CK may also have transformations (see Section 2.1.3.2).

[4] http://github.com/spgroup/theory-pl-refinement/tree/dev

Instead of defining similar templates just differing on the CK language used, we ideally would define a single template that is more general and compatible to both languages. This is impossible because in all scenarios presented here, the CK may be modified. So, we need to show such modifications and we can only choose to use one of the languages. For example, in the REMOVE FEATURE template, the compositional version assumes that the CK has feature expressions and asset names, whereas the transformational version deals with transformations instead of simply names. They also differ in the structures underlying their similar syntax CK. While the former definition consists of a set of items, the latter conists of a list. It is not feasible to work with these different representations at the same time. Additionally, a number of conditions may vary and impose restrictions on the artifacts format. For instance, one could assume an artifact with an *ifdef* block, while another simply does not deal with such structure. For these reasons, we present two versions of the REMOVE FEATURE and other templates.

Remove Feature

Developers may need to remove features for diverse reasons, as already explained in Section 4.1. Thus, we also have a template to deal with feature removals but allowing the use of transformations in the CK and *ifdefs* in the AM structure. We could also have another template considering the use of the *select* transformation, which would be similar to the compositional one, as *select* transformations do not really transform assets, but simply select them. For this reason, we only present a template dealing with *ifdefs*. In Figure 4.5, it is possible to notice that the three parts of the product line are affected. Similarly to the feature removal template for compositional CKs, we give support for the products that do not have the removed feature. So, S is defined in the same way.

The O feature is removed from the initial FM F. Similarly to its compositional version, assets implementing the O feature should be removed from the product line. In this case, since we are able to transform assets, we can use preprocessing directives to implement features. So, the x tag activates the c code that implements O. Thus, in the new AM, this tag is not present, neither the c code. Two lines are removed from the initial CK. Both have an expression e, which, if true, implies in the presence of O. For this reason, both lines should be removed, otherwise we could have a malformed CK, that refers to features that do not exist anymore. The first transformation in the first line is *tag* x, which activates this tag. So, the transformation *preprocess* n generates a new asset with the code respective to the activated tags. As a consequence, since the x tag is previously activated, c is included. If x had not been activated, c would not be included.

There are three other conditions in the template illustrated in Figure 4.5. The first one is to make sure that O only appers in e. This is essential to guarantee that the remaining expressions in the resulting CK will not refer to a feature that was already

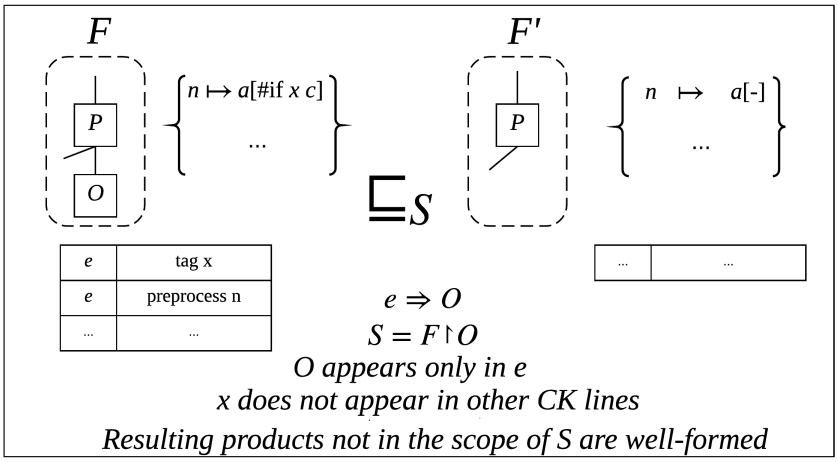

Figure 4.5 – REMOVE FEATURE partial refinement transformational template

removed. We also require the x tag to not appear in other CK lines. Differently from the other conditions, we can not find this one in the compositional REMOVE FEATURE template, as we do not deal with tags there. This is necessary because since this tag activates the code related to the removed feature O, it is also removed. The first CK line refers to x, and it is also removed. So, since we are assuming that x refers to the removed feature, it makes no sense to allow that other CK lines refer to x. This could imply in a illformed product line because after removing O, there would still be a tag referring to O. Finally, we need a well-formedness condition. We do not have control over the products that are not in the scope of S, since these products were affected with the feature removal. Consequently, we do not know if these products would still compile successfully, for example. So, we need to establish that they must be well-formed after the change.

We do not present the formalization of the REMOVE FEATURE transformational template, but it follows a similar reasoning to the REMOVE FEATURE compositional template proof illustrated in Section 4.1. Nevertheless, the languages used to represent the CK and assets are different because we allow CKs with transformations and *ifdef* blocks. Moreover, we prove the template shown in Figure 4.5 by induction over the CK, since we deal with a recusive semantics function. In contrast, the compositional template proof requires no induction.

Adding and Removing Assets

As we discussed in Section 4.1, developers might need to add new assets to a product line. We support them with the ADD ASSETS template that represents this scenario assuming a compositional CK structure. The template shown in Figure 4.2 is not compatible with transformational CKs because it deals with a CK that is a set of items, where each item has a feature expressions and a set of asset names. So, we now define an analogous template that allows the CK to be a list of items, with transformations. We can

guarantee that product configurations generated from F and do not satisfy expressions from its are refined. As we have that the evolved CK is $K +\!+ its$, this means that the new items need to be added at the end of the existing CK. Of course, if the new items are added to the beginning or in other positions, this should not make any difference, but formally these would be variations of this template.

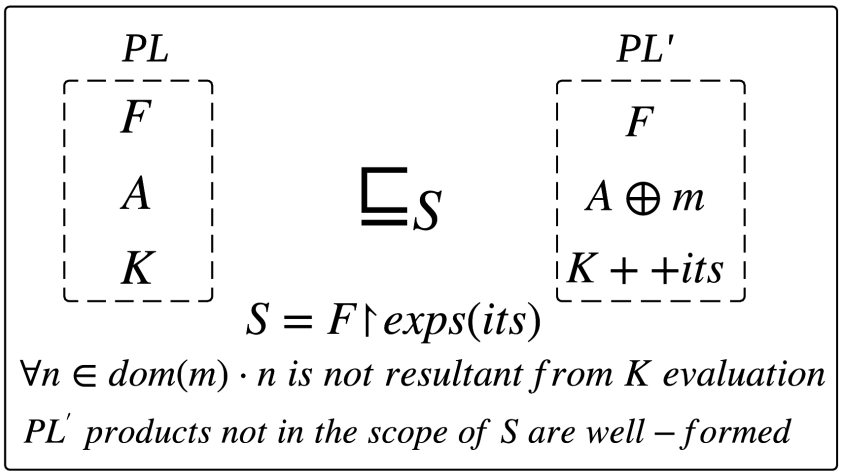

Figure 4.6 – ADD ASSETS partial refinement transformational template

There is a condition to make sure that assets from m do not appear in the original products. This condition is slightly different when compared to the compositional template, because it is not enough here to establish that assets from m do not appear in K. The reason is that K may have transformations, so the product is generated only after applying transformations. Thus, assets from m should not be found after transformations applied from K. As with every other non-refinement transformation involving assets, we can not guarantee that products not in the scope of S remain well-formed. So, as every product line is well-formed by definition, we need this condition to guarantee that the transformation will not break this condition.

We should observe that, similarly to other templates, the ADD ASSETS template can be derived by combining two other. Applying the ADDING UNUSED ASSETS [NBA+15] (refinement) template followed by the ADD CK LINES (partial refinement) results in the change described in Figure 4.2. This allows us to prove this template by reusing proofs from the two others. As a consequence of having derived templates by applying others sequentially, our set of templates is not minimal.

Similarly to the ADD ASSETS template for compositional CKs, if we read the template from Figure 4.6 from right to left (and considering the well-formedness condition for PL instead of PL'), we have a possible REMOVE ASSETS template for CKs with transformations. We do not present this template here for simplicity, but it is specified and proved in our online appendix.

Changing, Adding and Removing CK Lines

We have already explained this scenario in Section 4.1. However, the template in Figure 4.7 is not dealing with CKs with transformations. As it is possible to notice, it assumes that the CK is formed of feature expressions and asset names. The difference in this template from Figure 4.7 is that we use a slightly different notation for the CK that assumes that the CK is a list. This is essential for the CK with transformations, because the order in which the transformations appears matters. Additionally, instead of having asset names on the right side, it has transformations, like *tag* and *preprocess*.

Since we are also dealing with a change in a CK line, S does not change. It is defined in the same way, as the intersection of the configurations that do not satisfy e and e'.

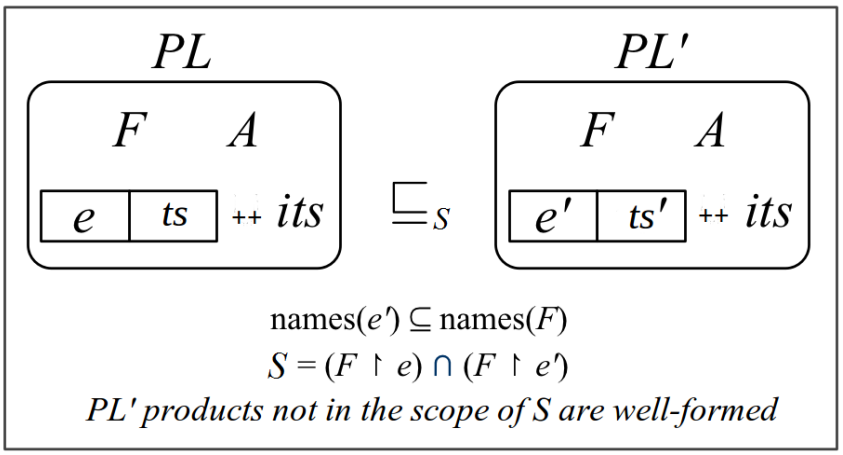

Figure 4.7 – CHANGE CK LINE partial refinement transformational template

Similarly to the templates presented in Section 4.1, we also have versions of the ADD CK LINES and REMOVE CK LINES templates for CKs with transformations. Visually, these templates are equal to the ones that are compatible with compositional CKs, but instead of the union operator, we need to use concatenation since transformational CKs are specified as lists. Due to this similarity, we do not show them here but we also support developers dealing with transformational CKs.

In this section, we present six partial refinement templates that deal with transformational CKs: REMOVE FEATURE, ADD ASSETS, REMOVE ASSETS, CHANGE CK LINE, ADD CK LINES and REMOVE CK LINES. We do not present any proof of the transformational partial refinement templates here, but they are available in our online appendix.[5] Compositional template proofs (like the one shown in Section 4.1) are more simple because the CK semantics function simply filters the assets that implement the selected features. In contrast, the transformational CK semantics function is recursive and assets are transformed with the use of *preprocess* commands, for example. So, the proofs

[5] http://github.com/spgroup/theory-pl-refinement/tree/dev

of the templates shown in this section were all made by induction over the CK items. This involves a higher complexity when compared to compositional template proofs.

4.3 General Templates

In this section, we present templates that do not require a particular CK language. This happens because in these scenarios, the CK does not change; only the FM or the AM. So, we can see the CK as a *black box*, we can abstract its structure. Consequently, these templates are compatible with both compositional and transformational CK notations. We first introduce the CHANGE ASSET template, which deals with changes only to implementation files. This template is a particular case of the AM partial refinement compositionality introduced in Section 3.2.2.2. Moreover, we also have templates to deal with changes only to the FM. These would be particular cases of the FM compositionality (see Section 3.2.2.1).

Change asset

Developers modify source files in many contexts, such as fixing bugs or implementing new features. In such situations, one possibly does not desire to preserve behavior. Thus, this is often an unsafe safe evolution scenario, since products that contain the changed asset might not preserve behavior. Therefore, we give refinement guarantees for the other products, which are the ones that do not have the changed assets. We define a template that matches this scenario in Figure 4.8.

To specify S for this case, we use another restriction operator. For arbitrary FM F, CK K and set of asset names ns, we use $(F, A, K) \upharpoonright ns$ to denote the subset of F configurations whose products (resulting from evaluating K with A and c) do not contain assets from ns. Hence, S is defined as $(F, A, K) \upharpoonright \{n\}$, which is the subset of F configurations whose features are not implemented by the asset named n, which in this case is the a asset. Since products containing a' are possibly not refined, we can not give any guarantees for them. There is also a well-formedness condition. Since we do not know which changes were performed to a, we need to demand well-formedness for products containing a'.

This template assumes that both product lines have the same F and K. More precisely, only the asset a is changed to a'. Thus, only the asset content is modified, not the asset name, which is the same for the initial and new lines (n). Although this template does not capture situations where the FM and CK change as well, one could obtain this effect by combining templates. The CHANGE ASSET template can be used with the CHANGE CK LINE template (which will be introduced later), for instance. Thus, developers could not only change assets, but also change their reference in the CK. As explained in Section 3.2,

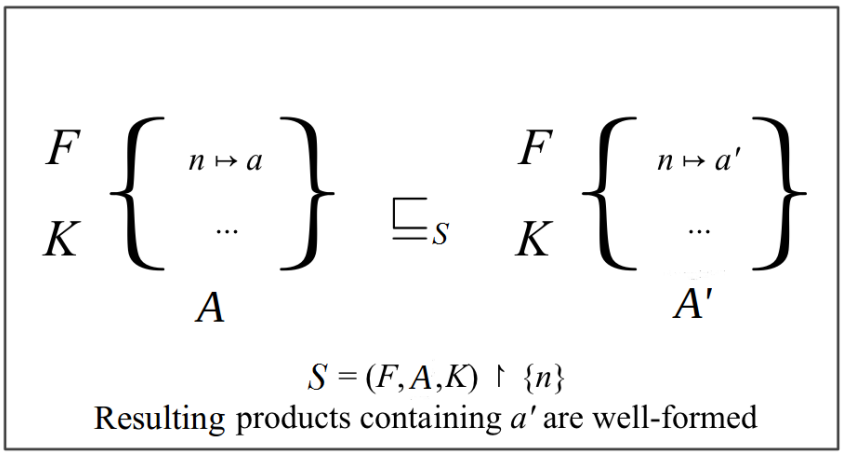

Figure 4.8 – CHANGE ASSET partial refinement template

the guarantee is for the intersection of the products refined in both steps and this can be automatically calculated, as we define S for both templates.

The CHANGE ASSET template also captures safe evolution scenarios. Suppose that one might refine an asset, this template also matches. However, we would give less support than possible since we assume that the asset is being changed in a non behavior-preserving way. The REFINE ASSET template [NBA+15] is more appropriate in this situation because it assumes that the product line is safely evolved and gives guarantees for all products. In contrast, if the change impacts the product line behavior, the REFINE ASSET template gives no support and developers should rather make use of the CHANGE ASSET template.

Transform Optional to Mandatory Feature

As already discussed in Section 2.1.1, features are classified into *mandatory*, *optional*, *alternative* and *or*. These types may change during the evolution process. Some of these changes are refinements and others are not. For example, transforming a mandatory feature into an optional one is often a refinement, since in the evolved product line we would have more configurations than before, but we would still have the existing products, and thus supporting existing users. This situation was addressed by previous work [NBA+15].

On the other hand, the opposite transformation, transforming an optional feature in mandatory, is often not a refinement. When we change a feature type from optional to mandatory, every product containing its parent then needs to have the changed feature. Thus, we would not be able to generate products without the feature. For this reason, some users would not be supported, but others can be because products containing the changed feature would be unnafected. We have a condition in the template (Figure 4.9) to guarantee that O can only be selected through the P selection, so there are no cross-tree constraints changing this condition. We state that we must be able to deduce the equation $O \Rightarrow P$ in CTC. We only have this type of restriction when the FM changes, so in the

other templates it is not necessary to check whether the cross-tree constraints satisfy a feature expression. So, it should not be possible to have O without P in the TRANSFORM OPTIONAL FEATURE TO MANDATORY template.

For this evolution scenario, we define S as the set of configurations that belong to the semantics of F, and satisfy the formula $O \vee \neg P$. This is expressed with the filter operator $\restriction$, which takes a FM F and a feature expression e and yields all configurations in F that satisfy e. This operator is the opposite of the restriction operator $\upharpoonright$. So, we give support for the original products that have O, because the only change applied to the initial product line was that O becomes mandatory. Furthermore, products without P are not affected, because they remain without O, as it is impossible to have O without having P.

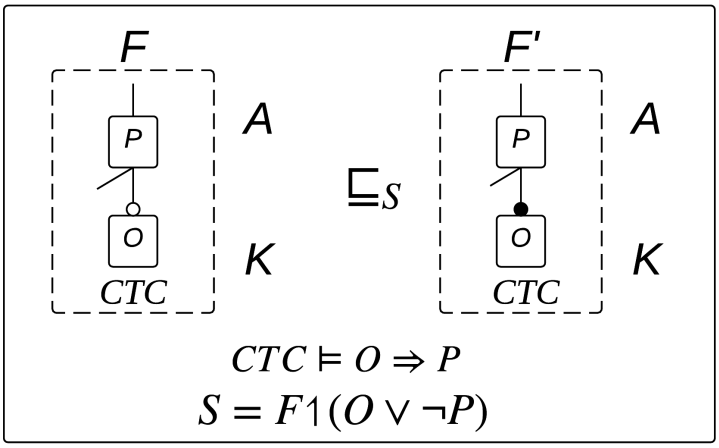

Figure 4.9 – TRANSFORM OPTIONAL FEATURE TO MANDATORY partial refinement template

We do not have any well-formedness condition for this template. This is not necessary because, in this particular case, we are able to prove that the resulting product line is well-formed. As there are no changes to assets in this case, we know that products remain well-formed. So, we had essentially to prove that all configurations belonging to F' semantics, also belong to F semantics.

Move Feature

Finally, we also consider changes to the FM regarding feature dependencies. During the evolution process, developers may want to move features. Moving only the code is safe, because this does not impact the behaviour of the feature. The only difference is where its code is located. Alternatively, features can be moved in the FM, and there are several possibilities. For example, a possible scenario is illustrated in Figure 4.10.

We have an initial FM F that has **at least** three features: P, Q and O. Feature P is the parent of Q and O. A change is performed and we then obtain F', where O is

now Q's descendant. In this scenario, product configurations from the initial product line that do not have Q and have O are inexistent in the resulting product line. In contrast, configurations that have O and Q do not suffer any impact, neither the ones that have P but do not have O. So, we define S as the set of configurations that belong to F semantics and do not satisfy the expression $\neg Q \wedge O$.

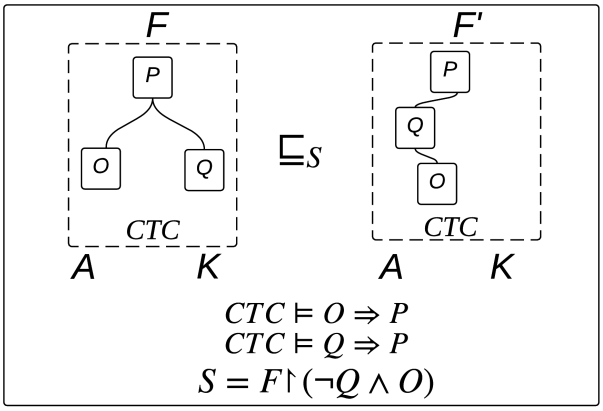

Figure 4.10 – Move feature to its sibling partial refinement template

The CTC meta-variable represents the cross-tree constraints and we require the initial FM to have the same constraints of the evolved FM. Besides that, the constraints should satisfy two expressions: $O \Rightarrow P$ and $Q \Rightarrow P$. This is necessary to guarantee that the cross-tree constraints are not changing the relation between the features. So, we should be able to select O only if P is selected and this must hold for the entire FM. The same happens to Q and P. So, both feature expressions should hold for the constraints of both FMs.

We do not specify any feature type in this template. This means that it is valid for situations where the features P, Q and O are of any type. However, depending on their types, we would be able to provide guarantees to a diferent set of products. For example, there are two situations where S is equal to all valid configurations, and as a consequence, we would have refinement. This happens when only Q is mandatory and when the three features are mandatory. In both cases, all products would have Q, so the expression $\neg Q \wedge O$, that should be satisfied by the configurations that are not refined, would not hold for any product. In any other scenario, S would not be equal to F semantics. For instance, if all features are optional, products containing only P and O would not be generated in the resulting product line, as Q would need to be present, since it is O's ascendant in F'.

We have just discussed a possible product line evolution scenario that consists of moving a feature in the FM, with the effect of changing feature dependencies. Nevertheless, as the FM structure is represented as a tree, there are several potential scenarios of moving features in the scope of the tree. We could, for example, have the opposite transformation, as illustrated in Figure 4.11.

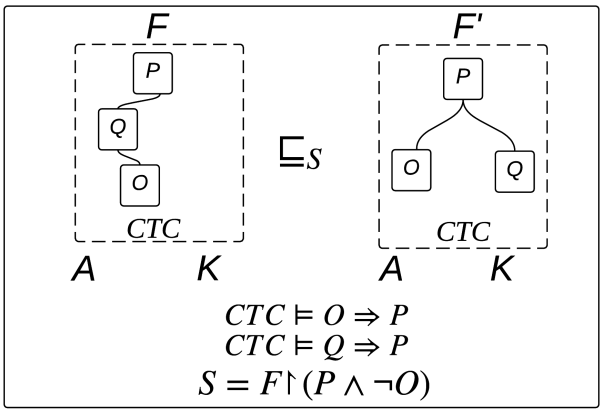

Figure 4.11 – MOVE FEATURE TO ITS PARENT partial refinement template

Again, only the FM is modified. The AM and CK remain the same. In the initial FM F, the O feature is Q's descendant, whereas in F' it is P's descendant. In this case, we do not support products that have P but do not have O. For example, the product containing only the P feature would not be refined if O is mandatory. So, we define S as the set of configurations that belong to F semantics, but do not satisfy the $P \wedge \neg O$ feature expression. As we opted to not specify feature types, the set of refined products depends on the type of O, if it is mandatory or optional. In the first case, the product containing only the feature P is only generated from the FM F, whereas in F' this is not possible. In contrast, if O is optional, this represents a refinement, because we are only increasing variability by allowing the generation of every initial product, and the product containing P and Q only.

Although we have shown two possible move feature transformations, there are several other possibilities. The FM tree can be large, and features may be moved to a place far from its origin. So, these are just examples and any case that do not match these templates needs to be analysed separately. Moreover, for each situation the value of the set of configurations S refined may vary. For this reason, we do not have a single template to represent all possible move feature scenarios.

We do not present proofs of the general templates, but they are available in our online appendix. The templates that deal with changes to the FM only (TRANSFORM OPTIONAL FEATURE TO MANDATORY, MOVE FEATURE TO ITS SIBLING and MOVE FEATURE TO ITS PARENT) are relatively simple to prove, as they do not deal with any change to the implementation, so we basically need to prove that the set S of product configurations refined can be obtained from the initial and evolved FM semantics. We also guarantee that the evolved FM is well-formed for these templates. We do not need to deal with CK semantics and AM peculiarities. Regarding the CHANGE ASSET template, it is a particular case of the AM partial refinement compositionality (see Section 3.2.2.2). So, we make use of the existing AM partial refinement notion to prove this template.

4.4 Template derivation process

We derived the templates by adapting a catalog of safe evolution templates [NBA+15, BTG12] for situations where not all products are refined by products in the evolved product line. For instance, the CHANGE ASSET template in Figure 4.8 essentially adapts the REFINE ASSET [NBA+15] template by dropping the precondition that the new asset a' must refine a. This way we allow any kind of change to a, but capture change impact by precisely defining S. Verifying completeness of the templates and proposing a minimal set are part of our future work. We would also possibly need more templates, but we already cover several situations, like feature removals, CK line additions and removals, changes to the implementation, among others.

4.5 PVS Encoding

We now present the theory hierarchy created in PVS to add the partial refinement notion. In Figure 4.12, it is possible to observe not only the theories presented in Chapter 3, but also the theories created due to the templates. We highlight in dark grey colour (white font) the template theories. Although we do not show the entire hierarchy here, all PVS files and proofs can be found online.[6] As we discussed, we have three template categories and they vary mainly according to the CK notation used. Consequently, we have three different templates theories. For each template developers should obey the syntactic and conditions predicates like the ones presented for the REMOVE FEATURE template in Section 4.1. We also determine, for each case, the S set of product configurations refined. In the following, we explain each template theory for partial refinement.

- **PartialRefTemplatesComp**: this theory comprises the templates proposed in Section 3.4. It uses the concrete notions for FM (**FeatureModel** theory) and CK (**CKComp** theory). Since we are defining partial refinement templates, these theories all import the **PartialRefinement** theory.
- **PartialRefTemplatesTrans**: this is analogous to **PartialRefTemplatesComp**, but it deals with transformational CKs and uses the **CKtrans** theory instead of **CKComp**.
- **PartialRefTemplatesFM**: this theory corresponds to the templates presented in Section 4.3, except the CHANGE ASSET template that is specified together with the AM compositionality theory. This template is in a separate place because it assumes an *eval* function (see Section 3.2.2.2), which is part of the CK semantics. All the other templates deal with changes to the FM only, so all specific cases of the FM

[6] All PVS files and proofs are available at http://github.com/spgroup/theory-pl-refinement/tree/dev

weaker equivalence compositionality are specified here. These templates assume a specific notation for the FM, that is structured as a tree. So, we use the concrete FM theory and the intermediate CK theory **CKint**, since the templates do not specify any CK language and would be compatible with any concrete CK that is a instantiation of **CKint**.

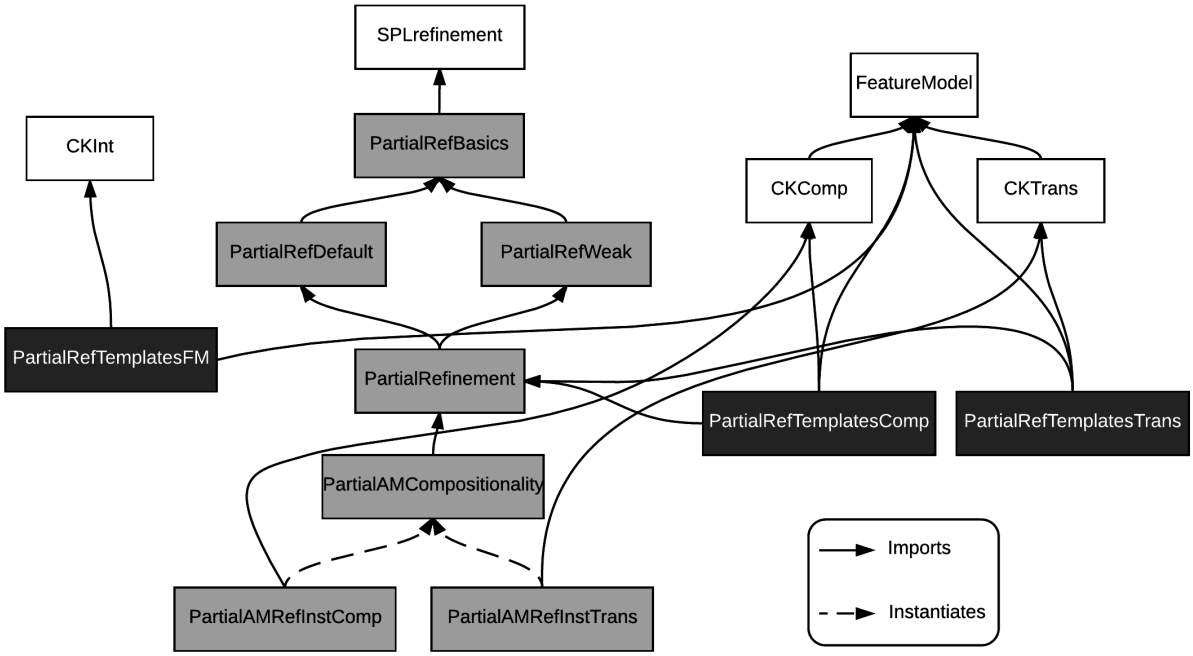

Figure 4.12 – PVS partial refinement theory

4.6 Discussion

In this chapter, we present partial refinement templates, that are divided into three categories: templates for compositional CKs (Section 4.1), transformational CKs (Section 4.2) and more general templates that assume a generic CK (Section 4.3). As discussed, these templates can vary according to the languages used to represent the three elements of a product line. We have two versions of each template that deals with a specific CK language.

Nevertheless, it is not our aim to present such catalog to developers. We would need to adapt the catalog for the specific needs and also take into consideration the languages being used. For example, if the company works only with compositional CKs, it would not make sense to include templates that deal with transformational CKs. Moreover, if the languages used are slightly different from ours, we would need to possibly redesign the existing templates. Finally, we could even consider evolution scenarios not considered yet, such as transforming an or feature group into an alternative one. All decisions depend on the user needs.

We should highlight that, in all situation analysed, products in the scope of S are actually equal in the two product lines. For example, in the compositional REMOVE FEATURE template illustrated in Figure 4.1, S is defined as the set of configurations that are in the initial FM and do not satisfy the O removed feature. Since the only modification in this scenario is associated to O, the other products are actually equal in both initial and evolved product lines. So, we could use the equality symbol $=_S$ symbol instead of the refinement one ($\sqsubseteq_S$). We should observe that, since refinement is a weaker relation than equality, in general, we are able to capture more scenarios with refinement and consequently provide a wider support for developers.

5 EVALUATION

Although we expect our partially safe evolution templates could be useful in a number of situations, it is important to gather empirical evidence so we can better understand how often they could be applied in practice. To do that, we perform a quantitative retrospective study by analyzing two product lines, Linux[1] and Soletta.[2] Both projects are active on GitHub, the variability model is written in Kconfig, Makefiles are used to map features to their implementation and *C* is the main programming language used for source code files. Linux is a large and highly variable system that has been used in previous works [IF10, ADSTDM08, DvDP16]. Soletta is smaller and more recent, so we also chose to analyse this system to understand whether characteristics such as project size and number of commits have any influence our analysis. Since Soletta's build system is based on Linux Kernel Kconfig, the configuration process is the same. Users can use the default configuration or a custom one, by choosing the features that will be present in the generated product.

We try to find scenarios that match our templates by analysing commits from the two projects. In this chapter, we detail the data extraction process in Section 5.1, show the results for each template in Section 5.2, and discuss threats to validity in Section 5.3. We structure the study using the GQM approach [Bas92], as we detail on the following.

Study main goal: The purpose of this study is to discover whether the proposed templates could be frequently applicable in a product line development context to reinforce their expressiveness. We would like to answer the following question:

Research question: How often would partially safe evolution templates be applicable in product line projects?

Metric: In order to answer this question, we automatically analyze commits from the Linux and Soletta projects, where each evolution scenario is composed of a commit pair: the initial and evolved product lines are the ones in two consecutive commits. We measure the number of occurrences of the proposed templates, since they represent partially safe evolution situations.

[1] Linux Kernel repository available at `http://github.com/torvalds/linux`
[2] Soletta project website: `http://solettaproject.org/`

5.1 Setup

We assess the templates using FEVER [DvDP16].[3] This tool, developed by Dintzner et al., is able to analyze commits from projects that use the Linux notation. FEVER takes a set of commits as input and collects all information from them. Then, the differences for each commit are processed and the resulting information is stored in a Neo4j database.[4] So, the tool discovers which Linux elements, such as the variability model, were changed in evolution scenarios. Additionally, changed files are automatically classified into source and non-source. To find occurrences of our templates, we query the database populated by FEVER filtering evolution scenarios by expressing the conditions for each template, such as whether they affect the FM.

We manually check all evolution scenarios to make sure they really match the templates (except those representing changes only to the implementation, as the number is extremely high), and also to reduce false positives and the tool imprecision. The analysis was made by one person only. So, we should also consider possible errors in this process. Altogether, we analyzed 67310 evolution scenarios of the Linux Kernel from the database we had access to, and this corresponds to all commits between Linux versions 3.11 and 3.16. The first commit was performed on September 2^{nd} of 2013 and the last one was on August 3^{rd} of 2014, so this comprises roughly one year of development. Additionally, we analysed 2300 commits of the Soletta project, from the project start, on June 26^{th} of 2015, until April 19^{th} of 2016, which corresponds to almost one year of development. We try to match each evolution scenario with the templates, based on their conditions, as explained in the following for each template.

Remove feature: scenarios that modify all three elements of the product line, removing elements. These three modifications must be correlated, as illustrated in Section 3.1. Thus, the removed mappings need to be associated with the removed feature in the FM. Similarly, the removed assets in the CK need also to be excluded from the implementation. These rules are detailed in Listing 5.1, using Neo4j query language. In the database, the *MappingEdit* and *FeatureEdit* nodes represent changes to the CK and FM, respectively. An *ArtefactEdit* is any file change. From the *MATCH* clause, we have all commits in which the CK and source code are both changed. We then have the *WHERE* clause to establish extra conditions. For instance, the first condition is that this commit should affect the FM as well, and the change must be a removal. Moreover, the feature name in the FM needs to be the same name as the edited feature in the mapping change (CK). Although the exact mapping between features and artifacts in Makefiles can be complex, FEVER relates each mapping change to one feature only, which may lead to imprecision. As the three parts are affected, we also state that there should be CK removals. It would make no sense to

[3] http://github.com/NZR/SPLR-FEVER-Tool.
[4] Neo4j website http://neo4j.com.

allow source code artifact additions in a feature removal scenario, so we filter these cases. We also verify if changes in the implementation are related to changes in the mapping. In the REMOVE FEATURE template, we have a condition regarding well-formedness that, in our analysis, we assume to be true in every scenario. All distinct commits obeying these rules are then returned.

Listing 5.1 – Remove feature Neo4j query

```
MATCH (file:ArtefactEdit)<--(c:commit)-->(mapping:MappingEdit)
WHERE
  (c)-->(:FeatureEdit{change:''Remove'',name:mapping.feature}) AND
  file.change=''REMOVED'' AND
  mapping.target_change=''REMOVED'' AND
  mapping.target_type=''COMPILATION_UNIT'' AND
  NOT (c)-->(:ArtefactEdit{type:''source'',change:''ADDED''}) AND
  file.name=~(''.*''+
              substr(mapping.target,0,length(mapping.target)-2) +
              ''.*'')
return distinct c
```

This query is subject to false positives. Since we are dealing with feature removals, it might be expected that source file modifications would be forbidden, like source file additions were. However, we should include examples with asset modifications because features are not necessarily implemented by one artifact only. So, when removing a feature, one can remove only an **#ifdef** block, without removing an entire file. Although we cover examples that deal with transformational CKs (see Section 2.1.3.2), this increases imprecision, since we can not filter if only an **#ifdef** has been removed. False negatives may arise due to special cases. For instance, the removed file not necessarily has the same name of the mapping target removed in the CK. Thus, this evolution scenario would not be found with this query because the last condition may not hold. Additionally, we also do not find scenarios that are compositions of feature removals and other changes. For instance, one could remove a feature and add a new one in the same commit. Errors in the dataset can also lead to false negatives.

CHANGE ASSET: we classified an evolution scenario as a change asset instance when only the implementation changed. In Listing 5.2, we show the CHANGE ASSET Neo4j query. We filter only commits that have at least one source file changed, as established in the *MATCH* clause. We also have some conditions in the *WHERE* clause. In the first two, we state that neither the FM nor the CK change. It was also necessary to establish that the commit had no added or removed source files. The last two conditions are useful to reinforce the first two. Therefore, we only capture cases where the change is in source code and non-code files, such as documentation. If only a *.txt* file is modified, we do not

consider it a change asset instance.

Listing 5.2 – Change asset Neo4j query

```
MATCH (c:commit)-->(a:ArtefactEdit{change :''MODIFIED'',
type :''source''})
WHERE
  NOT (:FeatureEdit)<--(c) AND
  NOT (:MappingEdit)<--(c) AND
  NOT (:ArtefactEdit
        {type:''source'',change :''ADDED''})<--(c) AND
  NOT (:ArtefactEdit
        {type:''source'',change:''REMOVED''})<--(c) AND
  NOT (c)-->(:ArtefactEdit {type :''build''}) AND
  NOT (c)-->(:ArtefactEdit {type :''vm''})
return distinct c
```

This query finds any scenario representing an implementation change. Nevertheless, we do not have a precise idea of the change type. A number of them might be refinement, and consequently instances of the REFINE ASSET template proposed in previous work [NBA+15]. We are not sure about the presence of false negatives, but they can be present due to dataset problems. For instance, if a *FeatureEdit* edge is added to a CHANGE ASSET commit; this way, we do not find it with our query. We do not show here queries for the other templates, but they follow a similar approach and are available in our online appendix.[5]

ADD ASSETS and REMOVE ASSETS: we classify evolution scenarios as instances of these templates when only the CK and implementation change. In the former, both changes must be additions. The files added to the implementation should be new and of type *source*, according to FEVER. For the REMOVE ASSETS template, the query is analogous. So, we only allow removals in the CK and implementation. Source files should be entirely removed and CK lines must also be excluded. Moreover, we have a similar condition to the last one in Listing 5.1 to guarantee that the changes are related. Developers might remove Makefile mappings and source files independently. So, we check whether the source file names appear in the affected CK lines. We do not consider any case in which the FM changes. This would be actually a feature addition or removal.

We are not aware of false positives that may arise due to the ADD ASSETS and REMOVE ASSETS queries. Regarding false negatives, we do not find instances in which an added file has not exactly the same name of the added CK line. However, changing such

[5] Our queries are available at http://github.com/spgroup/theory-pl-refinement/blob/dev/doc/evaluation/queries.pdf

condition would probably increase the false positives number.

CHANGE CK LINE, ADD CK LINES and REMOVE CK LINES: we identify these templates with only one query because they are very similar and we noticed that in some cases an evolution scenario was an instance of the CHANGE CK LINE template, but the Git diff algorithm was showing it as a removal followed by an addition. Since the tool relies on this classification, we could have non-precise results, so we preferred to detect mapping changes and check manually which templates match the respective evolution scenario. Another reason is that the number of instances is considerably small for these templates. For all of them, we required that the implementation and FM elements must remain unchanged. We also identified the other templates in a similar way, and the results are presented next.

We can have false positives in these instances because we are filtering any mapping change. So, a number of them might not be of our interest. We could filter mapping additions, removals and modifications separately, but the FEVER tool uses the Git diff algorithm, which has an unprecise classification. So, we prefer to filter all changes and manually classify according to their types. We are not aware of false negatives due to the query, but they can occur due to dataset problems.

5.2 Results

In this section, we discuss the results from the analysis of the Linux and Soletta systems. Linux and Soletta are similar with respect to the product line notation used, but vary in size and maturity level. So, these differences reflect in our results. For each project, we inform how often our templates could be applied in evolution scenarios. We also discuss threats to the validity of our results. Although we classify templates into compositional and transformational in Chapter 4, both systems analysed use a transformational CK notion. So, these results evaluate the transformational templates and evaluating the compositional templates is part of our future work.

5.2.1 Linux Kernel

First, we analyse the results of running FEVER against Linux commits. For each template, we discuss the number of instances found. We also provide examples of evolution scenarios that match our templates and show examples that were excluded due to problems related to query, dataset, among others.

Change Asset

According to Dintzner et al. [DvDP16], around 80% of feature oriented changes in Linux only touch the implementation, and do not affect the FM or CK. Confirming

that, the CHANGE ASSET template had the highest occurrence rate instances, with 55345 instances, which corresponds to almost 90% of the evolution scenarios analysed. This might be due to Linux maturity level, and also to the fine granularity of the commits observed in the analyzed period.

Asset refinements (REFINE ASSET template [NBA+15]) also match this pattern and, since the number of occurrences is extremely high, we could not manually verify all cases. Thus, a number of these occurrences might be full refinements. By manually analyzing 50 CHANGE ASSET instances (randomly chosen between versions 3.15 and 3.16), only 7 turned out to be asset refinements. The other 43 are non-refinements and the majority of them were bug fixes. Developers fixed such bugs, for instance, by modifying *if-then-else* conditions. Based on this analysis, we suspect that partial refinements occur more frequently, and this makes the CHANGE ASSET template far more frequently applicable than the REFINE ASSET template.

An example of a CHANGE ASSET scenario is the pair formed by commit *2627b7e15c*[6] and its predecessor. In Listing 5.3, a developer removes the call to the *ip_vs_conn_drop_conntrack* function (we used the − symbol to indicate the line removed) to avoid a crash, as he explains in the message. This is the only change; the other lines remain untouched. So, we consider this example to be a non-refinement, as there is a clear intention to change the feature behaviour by solving a bug. Moreover, regardless of the commit message, function call removals are often not refinement transformations [BSCC04, CCS10]. Unless the function has a void behaviour, the resulting program tends not to have a compatible behaviour to the initial one. In this example, products not containing the *net/netfilter/ipvs/ip_vs_conn.c* file are refined according to the set of products S specified in the CHANGE ASSET template.

Listing 5.3 – An excerpt of "net/netfilter/ipvs/ip_vs_conn.c"

```
if (cp->flags & IP_VS_CONN_F_NFCT) {
-    ip_vs_conn_drop_conntrack(cp);
/* Do not access conntracks during subsys cleanup because
nf_conntrack_find_get can not be used afterconntrack cleanup
for the net.*/
...
```

We do not know precisely the type of change performed in the 55345 commits returned by the query. A possible strategy could be processing the code changes to classify them according to the structures affected. For example, one might know if the change is a method addition, a class addition, a method call removal, among others. However,

[6] Change Asset example: http://github.com/torvalds/linux/commit/2627b7e15c. Rafael J. Wysocki commited on Jul 8, 2014; version v3.16-rc5.

this would require significant effort and we found no tool able to classify changes in *C* systems. So, we decided to follow a simpler approach, which analyzes commit messages for identifying terms that suggest that changes are not behavior preserving. It consists of analysing commit messages and ranking every term according to its frequency in the text. To perform this task, we performed the following steps: first, in *Neo4j*, we downloaded a *JSON* file containing all of the yielded information from the CHANGE ASSET template query. Thereafter, we developed a *Java* program that ignores information in the *JSON* file, like commit ids, authors, among others, except the messages themselves. This program then creates a text file for every commit message with the respective content. So, after performing this step, we had 55345 text files, each one containing the content of a commit message. The next step consists on using *Lucene*,[7] a natural language processing tool that analyses text files, and ranks every term found according to its frequency. Since every document here corresponds to a commit message, we are able to know the number of commit messages that had each term.

Lucene ranked 209.328 terms that were found in the 55345 messages. We can see from Table 5.1 that *Fix* and *Bug* occupy the 2nd and 146th positions, respectively. This suggests that a great number of the CHANGE ASSET instances are bug fix scenarios. *Patch* is the 3rd most used term, which also suggests the high presence of bug fixes or general improvements, which may or not be code refinements. Other words that might suggest the presence of product line refinement changes do not seem as frequent, like *Rename* and *Refactor*. This only gives a general idea, but we can not be sure about the exact changes performed in CHANGE ASSET without analysing the code. Messages may not be well-written, or incomplete. Developers often do not explain in detail their commits and surely express differently their ideas, so this is just an approximation. Being conservative and considering that only the documents containing *Bug* and *Fix* represent partial refinement scenarios and all the others are refinement, we then have 12680 CHANGE ASSET instances instead of 55345. The number decreases considerably, but we would still be able to support at 22% of the scenarios analysed.

The Lucene tool automatically excludes terms that are not of our interest, like prepositions, pronouns, among others. We also configured the tool to ignore others terms, such as *signed* and *off*, which are present at the end of every commit message and just pollute the rank, since they do not mean anything for us. Besides analysing the rank of every single term, Lucene also allows us to search for specific expressions, for example, the number of messages that contain *bug* and *fix*, and including other possible terminations, like *fixes*. This provides a more powerful search than the single term one, but there is no ranking in this case. We found similar results by looking for expressions involving terms, and the results are available in our website.

[7] Lucene website: http://lucene.apache.org/

Term	Frequency	Rank
Use	12609	1
Fix	11836	2
Patch	9921	3
Add	9916	4
Remove	8352	8
Error	4200	41
Change	4131	42
Bug	1870	146
Failure	1228	267
Rename	1111	305
Modify	431	954
Refactor	422	976

Table 5.1 – Frequent terms in CHANGE ASSET commit messages

In addition to analysing CHANGE ASSET commit messages, we also find commits that only change spacing in code files by using Conflicts Analyzer,[8] an open source tool that classifies conflicts according to a set of patterns [Acc15]. Although we are not dealing with conflicts, the tool identifies differences between source code files. So, for every CHANGE ASSET instance, we compare the initial and final files. From the 55345 commits returned in the CHANGE ASSET query, 777 only change whitespaces, so these can be considered product line refinements, which corresponds to approximately 1,4% of the instances for the CHANGE ASSET template. We consider this number to be high, but it depends on the project development practices. In the Linux case, changes have fine granularity, so this seems to be common. Commit *2055fb41ea*[9] is one of the instances found. In this commit, a line break is added before an *if* statement. No other changes are performed.

Our template could still be applied in this situation, because we do not make any restrictions to the changed artifact, but one should rather make use of the REFINE ASSET template, as it gives guarantees that all products are refined, differently from the CHANGE ASSET template, that assumes that the asset is not refined and gives behaviour preservation guarantees for only a subset of the existing products. For this reason, we exclude these instances. The other three excluded instances are permission changes. For example, commit *186026874c*[10] changes the permission code of a *C* source file from *755* to *644*. In the Git version control system, which we deal with, permission changes may be commited in projects whose configuration file has the *filemode* parameter set to *true*, like

[8] Conflicts Analyzer website: http://twiki.cin.ufpe.br/twiki/bin/view/SPG/ConflictPatterns

[9] http://github.com/torvalds/linux/commit/2055fb41ea. Rasmus Villemoes commited on Jun 20, 2014; version v3.16-rc3.

[10] http://github.com/torvalds/linux/commit/186026874c. Dave Airlie commited on Jul 2, 2014; version v3.16-rc4.

the Linux Kernel.

Adding, Removing and Changing CK Lines

There are at least 18, 9 and 12 scenarios respectively corresponding to mapping changes, additions and removals. We manually checked the 39 instances. There can be false negatives, for example, if the dataset is not correctly generated for some cases. In this case, our query would not return all examples. The numbers regarding these templates are not high because modifications focusing only on the mapping rarely occur, so the CHANGE CK LINE, ADD CK LINES and REMOVE CK LINES templates have a lower frequency when compared to others, such as ADD ASSETS. This might happen because most of the commits modify at least one source code file and some of them also modify the FM. The CHANGE CK LINE template presents the highest number of instances of the three patterns, probably because developers often remove and add mappings together with the respective source code associated or references to the FM as well. It is also possible that an evolution scenario captured by one of our templates corresponds to a longer sequence of commits. Since we try to match each commit pair separately with the templates, this would explain the low occurrence. We did not find any problem in the dataset, the main problem was query imprecision.

We provide an example of a CHANGE CK LINE instance in Listing 5.4, that shows the differences of *5a90af67c2*[11] according to its predecessor. The presence of the artifact *davinci-cpufreq.o* was conditioned to the activation of *CONFIG_ARCH_DAVINCI_DA850*. After the change, the *CONFIG_ARCH_DAVINCI* feature is mapped to this artifact instead. In the message, they explain that this commit fixes a build error. In such situations, there are no changes to the FM and implementation; only the CK changes, as we stated in our query. In this case, our template guarantees that products without the *CONFIG_ARCH_DAVINCI_DA850* and *CONFIG_ARCH_DAVINCI* features are refined.

Listing 5.4 – Changes made to "drivers/cpufreq/Makefile"

```
-obj-$(CONFIG_ARCH_DAVINCI_DA850) += davinci-cpufreq.o
+obj-$(CONFIG_ARCH_DAVINCI) += davinci-cpufreq.o
```

Each scenario is classified as compatible with one template only, except for the ADD CK LINES, CHANGE CK LINE and REMOVE CK LINES templates. Since they were found with the same query, we noticed that some scenarios actually had instances of more than one of the three patterns. Thus, a scenario might be classified as an instance of both

[11] CK line change commit: http://github.com/torvalds/linux/commit/5a90af67c2. Simon Horman commited on July 10, 2014; version 3.16

REMOVE CK LINES and ADD CK LINES templates, so we had to proceed with a manual analysis as follows.

Adding and Removing Assets

FEVER returned 181 instances of the ADD ASSETS template, which were all manually checked to confirm that they really match the template. Due to errors in the dataset, 13 of them did not. Therefore, we exclude these instances and only 168 remain, which precisely match our conditions and are instances of the template. There are at least 16 assets removals. We did not investigate the reason for such a lower removal rate. The results might be different considering another interval and project.

In Listings 5.5, 5.6 and 5.7, we show a scenario that matches the ADD ASSETS template.[12] Basically, a line is added to the Linux CK, to map the *CONFIG_SOC_EXYNOS5410* feature to the *exynos5410.o* asset. As this asset is new, the *clk-exynos5410.c* and *exynos5410.h* files are added to the implementation. So, as the CHANGE ASSET template requires, there is no change to the FM in this case. Also, the changes in the CK and implementation need to be related, and we do not allow source file removals or modifications. The only change in this commit that is not listed here is regarding documentation, but we do not forbid any change to a non-source file.

Listing 5.5 – Changes made to "drivers/clk/samsung/Makefile"

```
+obj-$(CONFIG_SOC_EXYNOS5410) += clk-exynos5410.o
```

Listing 5.6 – Changes made to "drivers/clk/samsung/clk-exynos5410.c "

```
209 lines added
```

Listing 5.7 – Changes made to "include/dt-bindings/clock/exynos5410.h "

```
33 lines added
```

As we have already discussed, we only classify as ADD ASSETS instances, commits that only touch the implementation and CK. So, other artifacts like the variability model are not allowed to change. However, among the commits returned by FEVER for our ADD ASSETS query, we found 1 commit that changes the Kbuild file, 11 that change the Kconfig and another example where additional CK lines change. In all these examples, there are additions to the Makefile mapping and implementation files. However, other changes are not allowed and should not have been returned by FEVER. For example, commit *d3e6573c48*[13] and its predecessor change additional lines in the CK. However, by

[12] Add assets commit: http://github.com/torvalds/linux/commit/e7ef0b632e. Kukjin Kim commited on May 26, 2014; version 3.16-rc1.

[13] http://github.com/torvalds/linux/commit/d3e6573c48. Haojian Zhuang commited on Dec 24, 2013; version v3.15-rc1.

a mistake, these extra changes do not appear in the dataset, so it would not be possible to filter them. The excluded instance of the REMOVE ASSETS template was commit *182f3fe929*[14] and its predecessor, which also changes the Kconfig. In this example, the dataset was generated without the edge that relates the commit and a Kconfig change. This should then be classified as a feature removal, where the three elements of the product line are affected. Although our query is correct and should not return such examples, the dataset was not correctly generated in a few cases.

Reason	# Excluded instances	Problem due to
Changes in Kbuild	1	Dataset
Changes in Kconfig	11	Dataset
Changes in additional CK lines	1	Dataset
Total	13	-

Table 5.2 – Excluded instances - ADD ASSETS template

Remove Feature

Our query returned 93 feature removal scenarios, but only 68 were classified as valid according to our templates. The other 25 non-removals, such as scenarios where the features were actually being moved. We did not consider these cases to match our pattern, since the feature was not actually removed. So, we exclude them. All excluded instances were found due to query imprecision, so we are not aware of bugs in the FEVER tool for this template. We already provide a valid example of the REMOVE FEATURE template instance in Section 3.1, and the examples are available in our website.[15]

We also present numbers regarding excluded instances of the REMOVE FEATURE template. In Table 5.3, we classify these scenarios into six categories. In five evolution scenarios of them, features were actually being moved/renamed. Commit messages help to identify these situations. In some examples, instead of being removed, features are simply being merged to other features. So, existing features incorporate their behaviour. For this reason, these instances were not considered feature removals. As we discuss, the REMOVE FEATURE template requires all parts of the product line to be affected. According to Table 5.3, only the FM was changed in four instances. In other commits, we found feature removals, but also extra changes, like dependency changes in the Kconfig. This would be the case of composing the REMOVE FEATURE template with a move feature template, for example. So, we also discarded one commit for this reason. Finally, we also observed that, in some examples, features were not being completely removed, and as this was not

[14] http://github.com/torvalds/linux/commit/182f3fe929. Malcolm Priestley commited on Feb 13, 2014; version v3.15-rc1.

[15] http://github.com/spgroup/theory-pl-refinement/tree/dev

explicitly stated in the messages, we also discarded seven instances for this reason and some more commits that include changes to other artifacts not related to the feature being removed.

Reason	# Excluded instances	Problem due to
Move/Renaming	5	Query
Feature merge	3	Query
Changes in FM only	4	Query
Dependency changes	1	Query
Feature not completely removed	7	Query
Other	5	Query
Total	25	-

Table 5.3 – Excluded instances - REMOVE FEATURE template

Differently from the ADD ASSETS and REMOVE ASSETS problematic instances, REMOVE FEATURE ones were not returned due to dataset problems. They all satisfied conditions specified in the REMOVE FEATURE query, but the query is not precise enough to eliminate them automatically. One of the main problems is that we do not restrict source file modifications to deletions. Intuitively, one might argue that when a feature is removed from a product line, and no other changes are performed, we should have source file deletions, but not additions/modifications. This would be valid in a compositional product line development context, where one code artifact implements exclusively one feature. However, in the Linux system, developers can make use of *ifdefs*, so an artifact may be the implementation of more than one feature, and removing a feature from the code means basically removing the respective *ifdef* only. For this reason, we allow source code removals and modifications, but we need to filter them manually.

Summary

Now, we present the numbers of each template in Table 5.4. As discussed, the CHANGE ASSET template had the highest number, corresponding to almost 90% of the evolution scenarios analysed. By identifying commits that actually change only whitespaces and permissions, we exclude 780 scenarios. We try to reduce the query imprecision by analysing commit messages, which suggest that non-refinement occur more frequently than refinement scenarios. The other templates had considerably low numbers. The ADD ASSETS query returned 181 instances, but, as explained, only 168 were classified as valid, since the other 13 had unnexpected characteristics due to dataset and query problems. Similarly, the REMOVE ASSETS template had 16 instances only. The templates that deal with changes to the CK (CHANGE CK LINE, ADD CK LINES and REMOVE CK LINES) had 18, 9 and 12 instances. Although we do not exclude any scenario, we believe that having only one query to find three templates can lead to imprecision. Finally, the REMOVE

FEATURE query returned 93 instances and after manual analysis 68 remain valid. We believe that there are no problems due to dataset in this template, but the query is not precise enough.

Template	Query returned	Excluded	Problems	Remaining
CHANGE ASSET (and possibly REFINE ASSET)	55345	780	Changes can be of any type	54565 (89.4%)
ADD ASSETS	181	13	Dataset	168 (0.27%)
REMOVE ASSETS	17	1	Dataset	16 (0.02%)
CHANGE CK LINE	18	0	Query	18 (0.02%)
ADD CK LINES	9	0	Query	9 (0.01%)
REMOVE CK LINES	12	0	Query	12 (0.02%)
REMOVE FEATURE	93	25	Query	68 (0.11%)

Table 5.4 – Template occurrence - Linux Kernel

As explained, the numbers in Table 5.4 are lower bounds of the cases we could confirm. We provide a summary of our analysis in Table 5.5. From the 67310 commits, 5413 are merge commits, which are discarded by the tool because they correspond to integration, not evolution, scenarios. Although there might be changes during manual merges, they are not really relative to a single previous product line, as captured by our templates. Hence, we could give support for approximately 90% of the cases. There are, in fact, 6221 commits that, together with its previous commits, do not match any of our current templates, which could include, for instance, commits that only change feature dependencies in the FM, or commits that represent feature additions, or even refinement scenarios such as feature renamings. As discussed in Chapter 4, the proposed templates were adapted from product line refinement templates proposed in previous works [NBA+15]. So, we aim to investigate these 6221 instances in more detail, and, if necessary, propose new templates to deal with them as well.

Total number of commits	67310
Merge commits	5413
Number of commits analysed	61897
Match our templates	55676 (89.94%)
Do not match any template	6221 (10.06%)

Table 5.5 – Template occurrence summary - Linux Kernel

We are not able to identify instances of all our current templates with the FEVER tool. For example, we do not provide any result for the TRANSFORM OPTIONAL FEATURE TO MANDATORY template. As we illustrate in Section 2.3, the Kconfig model does not provide a clear feature classification into *optional*, *mandatory*, *alternative* and *or*. So, the

current version of the tool is not able to inform a feature type. This would require a deeper interpretation of the Kconfig model, and possibly add and improve modules of the FEVER tool.

5.2.2 Soletta

Soletta is a development framework that makes writing software for IoT (Internet of Things) devices easier. With the significant recent development in this field, we have more connected devices, like coffee machines, printers, personal robots, among others. So, IoT is a growing area, and several applications are being developed in this context, including Soletta. By abstracting hardware and operating system details from a program, Soletta allows developers to write software for controlling actuators and sensors and communicating using standard technologies.

The same process used in Linux to find template instances was also applied to Soletta. By running FEVER and executing the queries, like the REMOVE FEATURE query detailed in Listing 5.1, we obtain the numbers regarding the Soletta project for the period ranging from 26 Jun 2015 to 19 Apr 2016. This interval corresponds to the whole history of this project up until the last analyzed commit. We analyzed 2300 commit pairs for the Soletta project.

In Table 5.6, we notice that the numbers are significantly lower when compared to Linux. This was already expected, as this project is considerably smaller and we are analyzing 2300 commits in this project. However, we can see some differences. The CHANGE ASSET query returned 1496 instances, or 65% approximately. In the Linux project, this template corresponds to almost 90% of the commits. We believe that this difference is due to the commits granularity and project's maturity level. In the Linux project, commits have a finer granularity than in Soletta, so developers commit more often in the former system. So, it is expected a higher CHANGE ASSET instances number. Furthermore, Linux is considered a stable project, so changes are performed mostly to the code, and there are less feature additions, for example, than a more recent project like Soletta.

We only found five ADD ASSETS instances. We expected to find more, as in the beginning the project might have a significant number of asset additions. Nevertheless, most asset additions are also feature addition scenarios, where, apart from CK and implementation, the FM also changes. So, we consider this hypothesis. These instances would best match the ADD NEW OPTIONAL FEATURE refinement template proposed in previous work [PTD+15]. There was no asset removal that matched our REMOVE ASSETS template. This is understandable because Soletta is relatively new.

The numbers for CHANGE CK LINE and ADD CK LINES were proportionally higher than in Linux. A possible explanation is that approximately 90% of the commits in Linux

only change the implementation, whereas in Soletta only 65%. Like REMOVE ASSETS, REMOVE CK LINES had 0 instances. We found five instances of the REMOVE FEATURE, but three were excluded. So, only two remain. This is also justifiable by the fact that this is a recent project, so we expected to find a greater number of additions instead of removals.

Template	Query returned	Excluded	Problems	Remaining
CHANGE ASSET (and possibly REFINE ASSET)	1496	0	Changes can be of any type	1496 (65%)
ADD ASSETS	5	0	Dataset	5 (0.22%)
REMOVE ASSETS	0	0	Dataset	0 (0%)
CHANGE CK LINE	9	0	Query	9 (0.39%)
ADD CK LINES	3	0	Query	3 (0.13%)
REMOVE CK LINES	0	0	Query	0 (0%)
REMOVE FEATURE	5	3	Query	2 (0.09%)

Table 5.6 – Template occurrence - Soletta

The results are summarized in Table 5.7. Surprisingly, we found only one merge commit for this period in Soletta. Thus, actually 2299 commits were analysed. As we explained, the FEVER tool ignores merge commits. So, at least 65.89% of the evolution scenarios would match our templates. This rate is much lower than Linux, that is almost 90%. We believe that this difference is due to the projects maturity level. The Linux project is older and the analysed interval in Soletta includes the start of the project, that tends to have more feature additions, which would not match any of our templates. Another possibility is the granularity level for the changes. Linux commits have a finer granularity. So, each commit in Soletta possibly would be the result of applying more than one partially safe evolution template. Since we try to match each commit pair to a single template, we would not include such instance.

Total number of commits	2300
Merge commits	1
Number of commits analysed	2299
Match our templates	1515 (65.89%)
Not match any template	785 (34.14%)

Table 5.7 – Template occurrence summary - Soletta

We have the same problems in Soletta and Linux regarding the results, as we use the same queries and FEVER in both projects. CHANGE ASSET instances are the most risky, since we do not precisely analyse the performed changes. So, we do not know the number of refinements, for example. Although we did not find any bug in the dataset for

the ADD ASSETS and REMOVE ASSETS, we encountered such errors in Linux, so we do not eliminate this possibility. For the other four templates, the queries are not precise enough and we do not consider problems in the dataset because we did not find any of them.

We excluded three REMOVE FEATURE instances from Soletta results. Commit *5293f12e59*[16] was excluded because it is basically a renaming, where the *FLOW_NODE_TYPE_FREEGEOIP* feature is renamed to *FLOW_NODE_TYPE_LOCATION*. So, this example is not considered to be a feature removal. We had similar situations in commits *8d2e8aeb2c* and *446bc7e43c*. We can see removals, but the features are actually renamed into others.

By running the Conflicts Analyzer tool over Soletta CHANGE ASSET instances, we did not find any commit changing only whitespaces. This may be due to two reasons: the number of processed commits (2300) and the project development practices. It might be the case that in the Linux project, commits have finer granularity and developers accumulate more changes before commiting. We also perform the term analysis for commit messages in Soletta. Lucene found a total of 6028 terms in the 1496 messages. As shown in Table 5.8, the word *fix* occupies the top again, as the most used term, appearing in 356 documents. Other words found in the Linux analysis like *Add*, *Error*, *Remove*, *Bug* and *Refactoring* also appear, but in lower positions. Although this project is different, we can see some similarity to Linux rank. This result indicates that bug fixes occur in a significant frequency in both projects, which is possibly higher than refactoring scenarios.

Term	Frequency	Rank
Fix	356	1
Add	307	2
Error	95	22
Remove	78	40
Change	74	42
Bug	14	390
Failure	5	825
Modify	5	826
Refactoring	3	1152

Table 5.8 – Frequent terms in CHANGE ASSET commit messages

[16] http://github.com/solettaproject/soletta/commit/5293f12e59. Flavio Ceolin commited on Sep 10, 2015; version v1_beta4.

5.3 Threats to Validity

As this is a preliminary evaluation, in this section, we discuss internal, external and construct validity.

Construct: As already mentioned in Section 5.2, to find occurrences of the CHANGE ASSET template, we search for any change in the implementation and do not analyze which type of modification was performed, thus possibly also retrieving commits which actually represent occurrences of the REFINE ASSET template [NBA+15]. Although we manually examined 50 commits and performed analysis using both Lucene and Conflicts Analyzer, we can not generalize to all commits. Regarding the term frequency analysis, we are aware that it is superficial to make conclusions, specially considering that developers express differently their ideas. Moreover, we do not consider synonyms in that analysis, which could also lead to more precise results. Scenarios matching the other templates can be safe only in pathological cases, so we do not take them into consideration.

Internal: We should consider that the tool we use may have bugs. The dataset generated by FEVER can have extra or less edges than it should, and this implies directly in our results. For example, if in a REMOVE FEATURE scenario, FEVER does not capture that the three elements of the product line have been changed, our query will not return such scenario. We discussed such examples in the previous section. We manually analyzed several instances, so there is evidence that, except for CHANGE ASSET instances, most of the results are correct. We should remind that the manual analysis was performed by one person only and this is also error-prone.

We did not find false negatives, but if we could detect and reduce them, we would have even better results. False negatives rate also depends on the number of commits matched to an evolution scenario. We analyzed each commit separately. For instance, one could remove a feature in two parts: first, the FM and CK could be changed, and in the subsequent commit only the implementation would be removed. In this situation, these two commits would not match any template, although, in sequence, they constitute a feature removal scenario.

We also consider the queries precision as an internal threat to validity, since it is not trivial to precisely classify commit changes. Finally, we assume certain template conditions to be true, such as well-formedness. To reduce such imprecision we should use a strategy to verify well-formedness [BGM+16] to make sure that the template would be applicable. In our analysis, we make the open world assumption. Consequently, we analyse the scenarios locally instead of globally. Thus, this is also a threat. In systems such as Linux it is not trivial to analyse changes globally and this is also part of our future work.

Furthermore, we should provide the set of products refined in each scenario. This way, developers know, for instance, when considering the motivating example shown in

Section 3.1, the exact set of products that had the *LEDS_RENESAS_TPU*. As already discussed, if all products have this feature, we would have no products refined and this means an empty S. Consequently, developers would have no support. In contrast, if the feature is not present in a high number of products, the support tends to be much higher. So, this is also a threat and part of our future work.

External: We only examined Soletta and a small part of the Linux repository history. Hence, we can not generalize the result for other history periods or projects, which may have different development practices, such as commits with coarser granularity and different programming languages. Perhaps, if we analyze other projects, the CHANGE ASSET template could be used together with others, since one might change not only the implementation but also the FM and the CK in a single commit. However, as a consequence, other templates could have a higher rate of occurrence. Although we do not include other projects, we consider the Linux system significant because of its popularity and complexity.

6 CONCLUSIONS

In this work, we define partially safe evolution for product lines. The motivation of this work is that product line evolution is a challenging and complex task. It is important to give guarantees during this process, even for a product subset only. Especially in highly configurable systems like the Linux Kernel, there are thousands of possible valid configurations and predicting whether products have their behaviour preserved is often hard.

This problem had already being tackled in the scope of safe evolution scenarios. A product line is safely evolved when all existing products have their behaviour preserved. So, we potentially have new products, as long as we are still able to generate the existing ones. This concept is formalised through a product line refinement theory and it is applicable in several scenarios, like code refinements, such as function renamings. Refinement templates capture these scenarios and abstractedly represents them to guide developers and avoid them overly reasoning about the theory.

Nevertheless, developers eventually perform unsafe changes. For instance, feature removal scenarios are possibly non-refinements, unless the removed features have a void behaviour. Otherwise, products having this feature do not have their behaviour preserved. The existing theory does not give any support for this and other unsafe scenarios. We should notice, however, that the other products (the ones not containing the removed feature) are not affected. So, the evolution can be safe according to this specific products subset. This is the idea behind the partially safe evolution concept. We formalise this concept in the PVS theorem prover through a partial refinement theory. We use the S index to represent the set of products refined and make considerations regarding its size. We also define and prove a set of properties regarding partial refinement, like pre-order, which is essential to guarantee that every product line partially refines itself and after several partial refinement operations, the final product line partially refines the original one. We then show that we have actually two definitions of partial refinement, and the main difference is the form they handle feature names. (Definition 8) does not deal with feature renaming scenarios, so in this case one might need to use the weaker partial refinement notion (Definition 13).

As we have discussed, in a product line development context we might have safe and partially safe evolution scenarios, interchangeably. So, we also establish properties connecting these notions to suggest how they can be used together. One may need to perform, for instance, a feature removal followed by a function renaming and we also provide transitivity properties assuming that refinement and partial refinement operations

can be interleaved. Apart from transitivity properties, we have a commutable diagram, which indicates that we may perform refinement and partial refinement transformations and still achieve the same target. This shows that the order that the transformations are applied does not matter.

Regarding the partial refinement theory applicability, we define a template catalog that abstracts common partial evolution scenarios found in practice. For each template, we analyze the change impact and give refinement guarantees for a proper subset of the original products. A template describes how each product line element can be modified in an evolution scenario.

Templates can be more general than others, in the sense that they are compatible with several languages of the three elements. However, it is usually not possible to generalize compositional and transformational CK languages when we need to present the CK in detail. For this reason, we divided the templates into three categories: general, compositional CK and transformational CK. Compositional and transformational CK partial refinement templates are similar and represent the same scenarios, but proofs differ due to the semantics notion used. The entire partial refinement theory and the templates are mechanised in PVS. As our motivating example (illustrated in Section 3.1) shows a feature removal scenario found in the Linux Kernel commits history, we present the entire proof of the compositional Remove feature template. The other proofs are available in our online appendix.[1]

Finally, we gather template occurrence evidence in two product line projects: Linux and Soletta. We believe that these two open-source projects can reveal if the proposed templates might be useful. Moreover, they represent two opposite extremes in terms of size, number of contributors, number of features, among others. We used FEVER to analyse both projects. The tool takes a set of commits as input, and creates a Neo4j database informing which part of the product line was changed in the specific scenario. So, we create queries for our templates to find instances in the database. We analysed 67310 commits of the Linux system, which corresponds to all changes made from version 3.11 to version 3.16. We found that we could support approximately 90% of the evolution scenarios. More than 89% correspond to implementation changes only. So, we conclude that during this period Linux developers almost did not change the FM and CK elements and we speculate that this is due to Linux stability level. Consequently, the other templates, for instance Add assets, Remove feature and Change ck line, had low occurrences.

We found slightly different results by analysing 2300 Soletta commits. We could support approximately 65% of the evolution scenarios. We believe that since the Soletta project was created in 2015 and we analysed its start, developers might have added more features, for example. As we do not have any feature addition partial refinement template,

[1] http://github.com/spgroup/theory-pl-refinement/tree/dev

this type of scenario is not returned by our queries. Nevertheless, this could be the case of a product line refinement. So, some scenarios that do not match our templates could be supported by the refinement theory.

6.1 Contributions

This work has the main following contributions:

- A partially safe evolution concept for product lines that is formalised through a partial refinement theory (Chapter 3);
- A set of properties to support developers, for instance, in the step by step evolution and dealing with refinement and partial refinement theories interchangeably (Chapter 3);
- A template catalog that abstractedly represents partially safe evolution scenarios and precisely inform the subset of products refined in each situation (Chapter 4);
- A study to find evidence of the templates applicability in two product lines project: Linux and Soletta (Chapter 5).

6.2 Related work

This section presents related work in tools and formal verification for software product lines. We discuss the different approaches and similarities with this work.

As discussed, this work is based on previous works. Borba et al. [AGM+06, BTG12] define safe evolution for product lines. A product line is safely evolved when behaviour preservation holds for all initial products, and this is formalized through a refinement theory. Teixeira et al. [TBG15] extended this work for product populations and multi product lines. With the aim of guiding developers in possible refinement scenarios, Neves et al. [NBA+15] and Benbassat et al. [BBT16], among others, propose template catalogues to abstract safe evolution scenarios. Finally, a product line of theories for reasoning about safe evolution of product lines was proposed by Teixeira et al. [TABG15] to investigate and explore similarities between different languages that specify product line elements.

Dintzner et al. [DVDP14] present a classification of feature changes as well as a tool named *FMDiff* to automatically analyze differences in Linux variability models. The change categories are specific to structures found in Kconfig specifications, such as feature dependency changes. Finally, they evaluate the tool by analyzing commits from the Linux repository history. Thüm et al. [TBK09] classify FM edits into refactorings, specializations, generalizations and arbitrary edits by using satisfiability solvers. Our work

differs because it deals with arbitrary changes to any of the three product line elements. We have properties dealing with changes to the FM, so these in particular could be used in such tools, providing even more support for developers when making changes to the FM, by informing the subset of refined configurations in each case.

Passos et al. [PTD+15] propose a pattern catalog containing feature addition and removal templates applicable in the Linux context. The main difference from their patterns to ours is that they do not focus on giving guarantees for developers in partially safe evolution scenarios. Additionally, they present both refinement and potential non-refinement templates. To verify the scenarios occurrence in practice, they conducted an experiment by manually analyzing the Linux repository trying to find instances of their templates and discarded the ones that did not present a significant occurrence rate. While they focus on proposing templates not only representing refinement scenarios but also non-refinements, our aim is to propose a new partial refinement theory and partial refinement templates. They also suggest the need for a new theory to address non-refinement scenarios.

Also in the context of the Linux system, Ziegler et al. [ZRL16] present an approach to identify relationships between configuration options, which allows to discover source files that might be affected due to a change in a configuration option. They found that most configuration options affect few files only, and a few options affect a significant number of files. This work is related to ours, as we also analyse changes in the Linux system. However, this is not the core of our work. We could use their approach to present more detail of the evolution scenarios to give an idea of the number of products affected by a change to a configuration option. Furthermore, Lotufo et al. [LSB+10] provide a quantitative and qualitative analysis of the Linux product line. They discovered changes related to the FM, such as the number of features and the tree height, and how these changes influence in the Kconfig model complexity. While they focus on the Linux FM, we are interested in changes to the three elements of a product line.

Seidl et al. [SHA12] provide a remapping approach to keep product line artifacts after evolution. The authors classify changes to each product line element and inform developers possible inconsistencies that may arise. Our solution could be integrated to theirs in establishing other possible categories and supporting the inconsistency analysis, as we provide an impact analysis for a set of evolution scenarios.

Nieke et al. [NSS16] analyze feature model evolution and define temporal feature models, which allow features to have an expiration date. For instance, if a feature is removed it is no longer valid. It is also possible to have *locked* configurations. A configuration that is *locked* should never be broken. This information is achieved through analyzing possible changes, such as feature renaming, deletion, among others, to temporal FMs. This work resembles ours because it gives support for some partial refinements regarding the variability model. Developers can change some configurations and still be certified that

the *locked* ones remain valid. However, they only analyze the variability model and do not propose a partial refinement theory, differently from our work.

Also in the product line evolution context, R.P.d. Oliveira et al. [dOSdAdSG16] evaluates Lehman's laws of software evolution [Leh80] by checking their validity in two SPL industrial projects in the medical and financial domain. So, they analysed whether there is a relationship between six Lehman's laws and the evolution of common, variable and product-specific assets. Results shown that only one law was completely supported for all assets within both empirical studies. They then propose several guidelines to ease the product line evolution task, which is one of the main challenges in the SPL field. This work resembles ours because it aims to support developers in product line evolution, but the approaches are different. They try to check if Lehman's laws are valid in product line development contexts and suggest guidelines, and we tackle this problem in the context of partially safe changes and use formal reasoning.

Pfofe et al. [PTS+16] propose VariantSync, which is a tool for synchronizing software variants. The clone-and-own approach is often used by developers and variants may not evolve consistently. So, the gap to product lines tends to become larger over time. With this tool, one can reduce this gap by automatically synchronizing variants. The approach involves relating code changes to feature expressions. Thus, a code change can only be merged to a variant if the feature expression evaluates to true. This work is related to ours in the sense that they support developers in product line evolution, but whereas their strategy is tool-supported and focused on clone-and-own product lines, ours involves a refinement theory and focus on partially safe evolution scenarios. Theories that reason over product populations or multi-product lines might be useful in this context [TBG15].

A number of researchers [CHS+10, GLS08, LPT09, CCP+12] use model checking techniques [CGP99] to verify products lines. Sabouri and Khosravi [SK14] try to tackle the well-known state space explosion problem by statically analyzing product family models, before checking them against properties (expressed in linear temporal logic [Eme90]), to avoid re-verifying products by using previous results. This work is related to ours, since in both proposals, not all products are verified. However, we defined a partial refinement theory and verify product refinement, whereas their focus is not refinement, but to verify general properties, such as whether a product has a specific feature. There are also approaches to verify properties regarding probabilistic product lines. Chrszon et al. [CDKB16] defined the ProFeat language to specify probabilistic product lines and a translation engine from ProFeat models to PRISM [KNP02] models, so that they can use the checker to verify such properties. Again, the focus is not refinement, whereas we do not deal with probabilistic product lines.

Finally, there are several works [RST+04, ZKK12, Mon11] that propose change impact analyzers in a specific context, such as the Java language. The approaches involve

running tests to check whether behaviour is preserved after a change. Our work is also related to change impact analysis but we do not deal with any programming language in particular, so our discussion is more abstract. Furthermore, we reason about changes not only to code, but also FMs and CKs, as we are dealing with product lines.

6.3 Future work

As we mention in Chapter 3, we reason over scenarios that combine refinement and partial refinement operations, as they may both occur in practice. In this context, we show that performing these operations in different orders can lead to the same resulting product line. We have a property relating partial refinement and name aware refinement, but we also intend to expand our theory to deal with function transformations to specify refinement (see Section 3.3), and also prove that refinement and partial refinement commute. Additionally, we intend to correlate our work with previous work [TABG15] that defines a product line of product line theories. This way, we could apply our theory to itself and integrate our new theory to the existing product line of theories.

We have defined a number of templates, but there are potential partially safe changes that we are not dealing with. For example, transforming an OR group into an alternative group would be a non-refinement scenario, since we are not able to select more than one feature in the alternative group of the evolved FM. We also intend to investigate variations of the proposed templates. For instance, the Remove feature template assumes that the sub-tree under the removed feature is also removed. Thus, this does not allow developers to remove a feature in the middle of a FM keeping its children. So, we could have more templates to deal with this and other partial refinement situations. Although we have templates to deal with different CK notions, we do not investigate the compatibility of our templates regarding more complex FMs that deal with cardinality and attributes, for example. This is also another possible future work.

We intend to expand our empirical analysis, which is still preliminary, by evaluating other projects, and determining the exact value of S for each scenario. If so, developers would be aware of the exact set of products refined in each evolution scenario. We would then have a better idea of the support provided. We could also propose a minimal set of templates and investigate their completeness. If it is not possible to obtain absolute completeness, we could then establish a relative completeness by showing that the templates are expressive enough to transform an arbitrary product line to a reduced normal form. Furthermore, we plan to answer other research questions, such as if our approach is safe and also check the number of developers that performed safe and unsafe changes. Interviewing developers and using bug trackers to understand if there are bugs related to safe scenarios are also part of our future work.

Additionally, we could also further investigate CHANGE ASSET instances. Although we provide a commit messages analysis, it would be useful to know precisely the type of changes in each scenario, and also classify them in refinements and non-refinements. A possible approach is to use the SafeRefactor tool [SGSM10, MRG+17] for C programs to identify scenarios that are actually refactorings. We should consider verifying product line well-formedness considering configurable systems [MRG+17, BGM+16]. We also intend to classify refactorings into root-canal or floss [MHPB12], and give a precise difference of number of *ifdefs* during evolution scenarios. We could also analyse, for each CHANGE ASSET instance, the number of code lines changed. We believe that this number is considerably low in Linux because commits have a fine granularity. The FEVER tool can also be used to discover the refinement templates instances and one could compare the results with partial refinement instances. Moreover, we also intend to perform a study where developers apply the templates in real time. This could lead to different results from our analysis, which is retrospective.

Although we defined partial refinement in the product line context, we are not aware of such notion for arbitrary programs. So, another possible future work is define partial refinement for sequential programs, for example. One would then build on the traditional refinement notion for sequential programs, that consists on weakening preconditions and strenghtening postconditions. An analogous reasoning could be applied to concurrent programs. A partial program refinement notion would allow us to increase the support in some scenarios. Instead of giving support for a subset of products only, for the affected products, we would give support for part of the product.

Another possible expansion of this work is dealing with probabilistic product lines, that are usually found in domains that include safety critical systems, such as avionics and automotive. This would involve defining refinement and potentially also partial refinement for probabilistic product lines. The ProFeat language already allows developers to specify these product lines and the models are verified using the PRISM checker. So, this could be extended with refinement.

Finally, we would like to develop a tool to support developers on software product line evolution. We could implement transformations described in the proposed templates to allow developers to automatically evolve product lines and inform the set of products refined. So, the tool would provide another layer of abstraction and use the partial refinement theory concepts in background. Furthermore, we also intend to provide a PVS guide with good practices. In total, we proved 54 theorems, 57 TTCs and 14 lemmas. So, we could share our experience using this theorem prover.

REFERENCES

[ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-oriented Software Product Lines: Concepts and implementation*. Springer, 2013.

[Acc15] Paola Accioly. Understanding conflicts arising from collaborative development. *International Conference on Software Engineering*, pages 775–777, 2015.

[ADSTDM08] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. The evolution of the linux build system. *Electronic Communications of the EASST*, 8, 2008.

[AGM+06] Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos Lucena. Refactoring product lines. *The International Conference on Generative Programming: Concepts & Experiences*, pages 201–210, 2006.

[Bas92] Victor R Basili. Software modeling and measurement: the goal/question/-metric paradigm. Technical report, 1992.

[Bat04] Don Batory. Feature-oriented programming and the ahead tool suite. *International Conference on Software Engineering*, pages 702–703, 2004.

[BBT16] Fernando Benbassat, Paulo Borba, and Leopoldo Teixeira. Safe evolution of software product lines: Feature extraction scenarios. *The Brazilian Symposium on Software Components, Architectures and Reuse*, pages 11–20, 2016.

[BGM+16] Larissa Braz, Rohit Gheyi, Melina Mongiovi, Márcio Ribeiro, Flávio Medeiros, and Leopoldo Teixeira. A change-centric approach to compile configurable systems with# ifdefs. *International Conference on Generative Programming: Concepts & Experiences*, pages 109–119, 2016.

[BSCC04] Paulo Borba, Augusto Sampaio, Ana Cavalcanti, and Márcio Cornélio. Algebraic reasoning for object-oriented programming. *Science of Computer Programming*, 52(1):53–100, 2004.

[BTG12] Paulo Borba, Leopoldo Teixeira, and Rohit Gheyi. A Theory of Software Product Line Refinement. *Theoretical Computer Science*, 455:2–30, 2012.

[CCP+12] Maxime Cordy, Andreas Classen, Gilles Perrouin, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Simulation-based abstractions for software product-line model checking. *International Conference on Software Engineering*, pages 672–682, 2012.

[CCS10] Márcio Cornélio, Ana Cavalcanti, and Augusto Sampaio. Sound refactorings. *Science of Computer Programming*, 75(3):106–133, 2010.

[CDKB16] Philipp Chrszon, Clemens Dubslaff, Sascha Klüppelholz, and Christel Baier. Family-based modeling and analysis for probabilistic systems–featuring profeat. *International Conference on Fundamental Approaches to Software Engineering*, pages 287–304, 2016.

[CGP99] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[CHS+10] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. *International Conference on Software Engineering*, pages 335–344, 2010.

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

[dOSdAdSG16] Raphael Pereira de Oliveira, Alcemir Rodrigues Santos, Eduardo Santana de Almeida, and Gecynalda Soares da Silva Gomes. Evaluating Lehman's laws of software evolution within software product lines industrial projects. *Journal of Systems and Software*, 2016.

[DVDP14] Nicolas Dintzner, Arie Van Deursen, and Martin Pinzger. Extracting feature model changes from the Linux kernel using FMDiff. *International Workshop on Variability Modelling of Software-intensive Systems*, pages 22:1–22:8, 2014.

[DvDP16] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. FEVER: Extracting Feature-oriented Changes from Commits. *International Conference on Mining Software Repositories*, pages 85–96, 2016.

[Eme90] E Allen Emerson. Temporal and modal logic. *Handbook of Theoretical Computer Science, Formal Models and Sematics*, B(1072):5, 1990.

[FCS+08] Eduardo Figueiredo, Nelio Cacho, Claudio Sant'Anna, Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sérgio Soares, Fabiano Ferrari, Safoora Khan, Fernando Castor Filho, et al. Evolving software product lines

with aspects. *International Conference on Software Engineering*, pages 261–270, 2008.

[GLS08] Alexander Gruler, Martin Leucker, and Kathrin Scheidemann. Modeling and model checking software product lines. *International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 113–131, 2008.

[GMB05] Rohit Gheyi, Tiago Massoni, and Paulo Borba. A rigorous approach for proving model refactorings. *International Conference on Automated software engineering*, pages 372–375, 2005.

[IF10] Ayelet Israeli and Dror G Feitelson. The linux kernel as a case study in software evolution. *Journal of Systems and Software*, 83(3):485–501, 2010.

[KCH+90] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document, 1990.

[KNP02] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism: Probabilistic symbolic model checker. *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 200–204, 2002.

[Leh80] Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[LPT09] Kim Lauenroth, Klaus Pohl, and Simon Toehning. Model checking of domain artifacts in product line engineering. *International Conference on Automated Software Engineering*, pages 269–280, 2009.

[LSB+10] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. Evolution of the linux kernel variability model. *International Systems and Software Product Line Conference*, pages 136–150, 2010.

[MHPB12] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012.

[Mon11] Melina Mongiovi. Safira: A tool for evaluating behavior preservation. *International conference companion on Object oriented programming systems languages and applications companion*, pages 213–214, 2011.

[MRG+17] Flavio Medeiros, Marcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kastner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. Discipline matters: Refactoring of preprocessor directives in the# ifdef hell. *IEEE Transactions on Software Engineering*, 2017.

[NBA+15] Laís Neves, Paulo Borba, Vander Alves, Lucinéia Turnes, Leopoldo Teixeira, Demostenes Sena, and Uirá Kulesza. Safe evolution templates for software product lines. *Journal of Systems and Software*, 106:42–58, 2015.

[NSS16] Michael Nieke, Christoph Seidl, and Sven Schuster. Guaranteeing Configuration Validity in Evolving Software Product Lines. *International Workshop on Variability Modelling of Software-intensive Systems*, pages 73–80, 2016.

[OSRSC24] Sam Owre, Natarajan Shankar, John M Rushby, and David WJ Stringer-Calvert. PVS Language Reference. *SRI International*, 2001. Version 2.4.

[PBvDL05] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. *Software product line engineering: foundations, principles and techniques*. Springer, 2005.

[PTD+15] Leonardo Passos, Leopoldo Teixeira, Nicolas Dintzner, Sven Apel, Andrzej Wąsowski, Krzysztof Czarnecki, Paulo Borba, and Jianmei Guo. Coevolution of variability models and related software artifacts. *Empirical Software Engineering*, 21(4):1744–1793, 2015.

[PTS+16] Tristan Pfofe, Thomas Thüm, Sandro Schulze, Wolfram Fenske, and Ina Schaefer. Synchronizing software variants with variantsync. *International Systems and Software Product Line Conference*, pages 329–332, 2016.

[RST+04] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of java programs. *ACM Sigplan Notices*, 39(10):432–448, 2004.

[SBB+10] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. *International Systems and Software Product Line Conference*, pages 77–91, 2010.

[SBT16] Gabriela Sampaio, Paulo Borba, and Leopoldo Teixeira. Partially safe evolution of software product lines. *International Systems and Software Product Line Conference*, pages 124–133, 2016.

[SGSM10] Gustavo Soares, Rohit Gheyi, Dalton Serey, and Tiago Massoni. Making program refactoring safer. *IEEE software*, 27(4):52–57, 2010.

[SHA12] Christoph Seidl, Florian Heidenreich, and Uwe Aßmann. Co-evolution of models and feature mapping in software product lines. *International Systems and Software Product Line Conference*, pages 76–85, 2012.

[SK14] Hamideh Sabouri and Ramtin Khosravi. Reducing the verification cost of evolving product families using static analysis techniques. *Science of Computer Programming*, 83:35–55, 2014.

[TABG15] Leopoldo Teixeira, Vander Alves, Paulo Borba, and Rohit Gheyi. A product line of theories for reasoning about safe evolution of product lines. *International Systems and Software Product Line Conference*, 2015.

[TBG15] Leopoldo Teixeira, Paulo Borba, and Rohit Gheyi. Safe evolution of product populations and multi product lines. *International Systems and Software Product Line Conference*, pages 171–175, 2015.

[TBK09] Thomas Thüm, Don Batory, and Christian Kästner. Reasoning about edits to feature models. *International Conference on Software Engineering*, pages 254–264, 2009.

[TH06] Bruce Tate and Curt Hibbs. *Ruby on Rails: Up and Running: Up and Running.* " O'Reilly Media, Inc.", 2006.

[ZKK12] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. Faulttracer: a change impact and regression fault analysis tool for evolving java programs. *International Symposium on the Foundations of Software Engineering*, page 40, 2012.

[ZRL16] Andreas Ziegler, Valentin Rothberg, and Daniel Lohmann. Analyzing the impact of feature changes in linux. *International Workshop on Variability Modelling of Software-intensive Systems*, pages 25–32, 2016.