Karine Galdino Maia Gomes

CHARACTERIZING SAFE AND PARTIALLY SAFE EVOLUTION SCENARIOS IN PRODUCT LINES: An Empirical Study

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

Recife
2019

Karine Galdino Maia Gomes

CHARACTERIZING SAFE AND PARTIALLY SAFE EVOLUTION SCENARIOS IN PRODUCT LINES: An Empirical Study

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Engenharia de Software

Orientador(a): Leopoldo Motta Teixeira

Recife
2019

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

G633c Gomes, Karine Galdino Maia

Characterizing safe and partially safe evolution scenarios in product lines: an empirical study / Karine Galdino Maia Gomes. – 2019.

68 f.: il., fig., tab.

Orientador: Leopoldo Motta Teixeira.

Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2019.

Inclui referências e apêndices.

1. Engenharia de software. 2. Linha de produto de software. I. Teixeira, Leopoldo Motta (orientador). II. Título.

005.1 CDD (23. ed.) UFPE- MEI 2019-049

Karine Galdino Maia Gomes

"Characterizing safe and partially safe evolution scenarios in product lines: An Empirical Study"

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: 28/02/2019.

To my family

ACKNOWLEDGEMENTS

A Deus, que sempre demonstra Sua graça e Seu amor em minha vida, nas vitórias ou nas adversidades, e que está sempre guiando meus caminhos, me ajudando a combater o bom combate.

Aos meus avós e às minhas irmãs, por todo apoio e torcida. Um agradecimento especial à Kathi, que sempre esteve tão entusiasmada com meus sucessos, paciente nos momentos em que a paciência me faltava (não foram poucos haha), além de ser uma ótima companhia diária.

À minha família, que sempre me apoia, vibra e torce por mim. Em especial, gostaria de agradecer à Tia Selma, que sempre me deu suporte para que eu pudesse me dedicar aos estudos. Além dos bons princípios e valores, com ela também aprendi que estudos e livros são mais importantes que futilidades. Serei eternamente grata!

Agradeço ao meu Professor e orientador, Leopoldo Teixeira, pelos ensinamentos, paciência e confiança. Sempre muito presente e disponível para me ajudar em todas as etapas do programa. Aqui deixo meu muito obrigada e minha admiração pelo grande profissional que ele já é, apesar do pouco tempo de caminhada.

Além disso, sou muito grata pela oportunidade de ter cursado o mestrado no Centro de Informática da UFPE, um lugar cheio de profissionais com um amor visceral por ensino e pesquisa que são fonte de inspiração para mim.

Aos meus amigos (Klissiomara, Leuson e Neto) e irmãos de pesquisa (Alex e Thayonara), agradeço imensamente pela parceria no dia a dia acadêmico.

Também agradeço aos meus amigos da igreja e do colégio, que foram o equilíbrio perfeito entre momentos de descontração e apoio nos períodos de foco. Vocês são demais!

Por fim, peço a Deus para enxergar beleza e gratidão nos próximos desafios, frustrações e conquistas.

ABSTRACT

Software Product Line (SPL) is a family of software products that share common and distinct assets providing, through reuse, a systematic way to generate similar products. In SPL, each one of their characteristics is represented as *feature*, and the set of those features and its dependencies are expressed as Feature Model. Feature Model (FM) with both Configuration Knowledge (CK) and Asset Mapping (AM) spaces represent a SPL. Each one of those spaces play a key role to provide reuse in SPL. In the same way as regular software systems, product lines often need to evolve, such as adding new features, improving the quality of existing products, or even fixing bugs. Previous works have classified product line evolution scenarios into safe or partially safe, depending on the number of products that have their behavior preserved after evolution. Both notions rely on refinement theories that enable us to derive transformation templates that abstract common evolution scenarios. However, most of the works related to such templates are focused on either safe or partially safe templates. Therefore, in this work we aim to characterize product line evolution as a whole, measuring to what extent the evolution history in safe compared to partially safe, to better understand how product lines evolve from their conception. We measure how often these templates happen using 2,300 commits from the Soletta Project, an open-source framework for Internet of Things. Through our analysis, we observe that most of the commits were categorized as templates (78.3%). Further we make an evaluation for remaining one commits which were not categorized as templates (21.7%). Thus, we extract certain information for each evolution scenario, such as spaces affect, kind of modification (change/add/removed), amount of files and so on. In Soletta, we observe that several commits classified as templates are represented by *change asset*. Further, for the remaining commits, we observe that most of the commits modifies both CK and AM spaces. In other hand, fewest evolution scenarios modifies FM and AM spaces at the same time. Further, we distribute changes through the contribution time (timeline) from all 2,300 commits. Finally, after classify some commits manually and others automatically as safe and partially safe evolution, we obtain that in Soletta 91.8% of changes is categorized as partially safe, and the 8.2% remaining ones are safe evolution scenarios.

Keywords: Software Product Line. Product Line Evolution. Refinement. Safe Evolution. Partially Safe Evolution.

RESUMO

Linha de Produto de Software (LPS) é uma família de *softwares* relacionados que compartilham características em comuns, e outras distintas, visando gerar produtos semelhantes de forma automática através do reuso. Em LPS, cada característica é conceituada como *feature*, e o conjunto dessas *features* e suas dependências são expressas em um modelo, conhecido como *Feature Model* (FM). De maneira geral, FM associado com os espaços *Configuration Knowledge* (CK) e *Asset Mapping* (AM) representam uma LPS, e cada um desempenha um papel para geração de produtos através do reuso. Da mesma forma que sistemas tradicionais precisam de manutenção, correção de bugs, ou até mesmo adição de novas funcionalidades, LPS também evolui. Contudo, evoluir LPS é um trabalho que exige cautela, pois até mesmo pequenas mudanças podem impactar vários produtos. Diante disso, alguns trabalhos investigaram evolução de LPS, visando entender e dar suporte aos desenvolvedores durante as mudanças. Dentre os trabalhos anteriores, surgiu a teoria de **evolução segura** (com base na teoria do Refinamento), que afirma que determinadas mudanças preservam o comportamento da LPS anterior. Portanto, renomear um arquivo, ou adicionar novas funcionalidades opcionais são exemplos de cenários de evolução segura. Entretanto, nem todas as mudanças se encaixam nesse conceito, então, surgiu a teoria de evolução **parcialmente segura**, que afirma que existe um subconjunto dos produtos manterá seu comportamento preservado. Com base nas teorias de evolução segura e parcialmente segura, trabalhos anteriores derivaram *templates* que são modelos que abstraem scenarios de evolução com o objetivo de dar suporte aos desenvolvedores. Na literatura, existem trabalhos que focam apenas em evolução segura ou parcialmente segura, mas não em ambos casos. Portanto, esse trabalho tem como objetivo caracterizar evolução de LPS de forma geral, classificando mudanças como seguras ou parcialmente seguras, visando entender o processo de evolução de uma LPS desde a sua concepção. Para isso, analisamos 2300 commits do projeto Soletta, um *framework* que facilita a criação de sistemas voltados para Internet das Coisas. Como resultado, observamos que a maioria dos commits foi classificado como template (78.3%). Para os commits restantes, fizemos uma categorização de acordo com o tipo de mudança e o espaços afetados (FM, CK ou AM). 24% dos commits restantes apresentam mudanças no CK e AM simultaneamente, e apenas 3% apresentam mudanças que só alteram o FM. Ao final, nossos resultados mostraram que cerca de 91% dos das evoluções são classificadas como parcialmente seguras, e o restante como cenários de evolução segura.

Palavras-chaves: Linha de Produto de Software. Evolução de Linha de Produto. Refinamento. Evolução Segura. Evolução Parcialmente Segura.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Developing personalized software often implies in high costs and efforts, but in other hand yields customization. Since software products share characteristics, rather than developing each one from scratch, it is more convenient to reuse components. However, the notion of reuse only reduces effort, but does not generate *personalized* products. In the context of software families, their products also share of a common set of characteristics, varying only in some aspects. This variation of some points enables customized products according to the customers needs.

Based on these issues, the Software Product Line (SPL) approach arises. SPL provides reuse and customization for generating a set of similar products in a systematic way. The notions of reuse and customization brings many advantages such as reduced costs, time to market, and quality improvements (POHL; BöCKLE; LINDEN, 2005).

SPL is usually represented as 3 spaces: Feature Model (FM), Asset Mapping (AM), and Configuration Knowledge (CK), and each one of these spaces play a key role to provide the reuse in SPL. In SPL, we use *features* to represent what might be variable or common among products. The set of all features and its dependencies are expressed in the FM (KANG et al., 1990). The FM is often expressed as a tree and contains a set of features names, and also documents variability, dependencies and its constraints (APEL et al., 2016). The constraints and dependencies among features allow generate a set of several valid products. Due to the FM being an abstraction of SPL domain, it is necessary to implement the features. Assets implement features and can be code, tests, documentation, or even images. Thus, the AM space constitutes a mapping from asset names to the actual assets. Finally, it is necessary to map features to their respective assets. Therefore, the CK plays a role in making this mapping, associating the features expressions with assets names. All of the three spaces are illustrated in Figure 1.

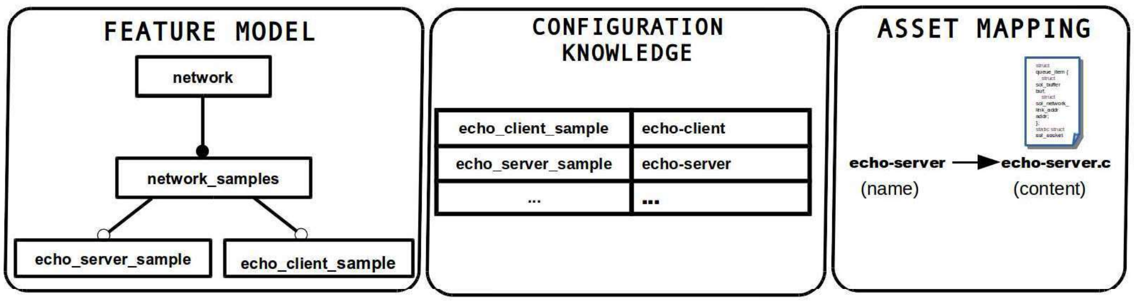


Figure 1 – Three Spaces of a Software Product Line

In the same way as regular software systems, SPL often need to evolve. So, adding new features, improving quality of existing products, or fixing bugs are common changes during development. However, evolution in SPL differs from single software products. Since a SPL consists of a set of reusable characteristics, one single feature could be present across a range of valid products. Therefore, evolution in SPL can be error-prone because a simple change can impact several products. Moreover, we need to take into account changes in the three spaces. Due to these error-prone changes, some research studies focus on evolution of product lines aiming to help developers during development to minimize the impact yielded by changes (LOTUFO et al., 2010; PASSOS; CZARNECKI; WASOWSKI, 2012; HEIDER et al., 2012).

The product line refinement theory (BORBA; TEIXEIRA; GHEYI, 2012) formalizes the notion of safe evolution, characterizing changes applied to SPLs that preserve the behavior of the existing products. Several changes performed by developers can be supported by this notion. For example, it is common to add an new optional feature, clean code removing unused assets, or even to refine some asset.

In fact, making safe changes is useful in SPL evolution, but there are also common changes not covered by this notion, since it requires behavior preservation of the entire set of existing products. For instance, during the life cycle of a project, it is common to remove some feature, fix bugs, or even change the implementation. However, these kind of changes may affect the behavior of some of the existing SPL products. Thus, the refinement theory was extended to introduce the notion of Partially Safe Evolution (SAMPAIO; BORBA; TEIXEIRA, 2016). This notion considers changes that preserve the behavior of only a subset of the existing products.

The theories that establish the notions of safe and partially safe evolution allow the derivation of transformation templates (BORBA; TEIXEIRA; GHEYI, 2012; NEVES et al., 2015a; BENBASSAT; BORBA; TEIXEIRA, 2016; SAMPAIO; BORBA; TEIXEIRA, 2016), that abstract a common evolution task, such as adding an optional feature. They also establish the necessary conditions for ensuring that the change is considered safe, or in the case of partially safe changes, the set of products whose behavior is unaffected by the change.

Existing studies only consider either safe (NEVES et al., 2015a; BENBASSAT; BORBA; TEIXEIRA, 2016) or partially safe evolution (SAMPAIO; BORBA; TEIXEIRA, 2016) scenarios, and do not examine the interplay between those two notions. Moreover, some studies only consider evolution of a single SPL space, such as variability model, for instance (KRÜGER et al., 2018; PASSOS; CZARNECKI, 2014). We aim to go beyond and analyze the SPL as a whole.

Therefore, this work improves the understanding about SPL evolution during its life cycle. This work aims to characterize product line evolution as a whole, measuring to which extent the evolution history is safe compared to partially safe. Our goal is to achieve a better understanding of SPL evolution, that might result in developing tools to assist developers on performing their changes.

This study then tackles two questions. First, *how are changes distributed in terms* *of safe and partially safe evolution during the SPL evolution history?* This might lead to patterns, such as performing safe changes more often during the project initial phases, and later mostly performing partially safe changes. Second, *how often do existing templates in the literature cover these evolution scenarios?* This question might serve as an assessment of previously proposed templates, and can also lead to deriving new templates.

In order to answer these questions, we realized an empirical study for analyzing a period of almost one year (between jun-2015 into apr-2016) of the Soletta project, totalizing 2,300 commits. Soletta is a real world and open-source development framework that makes writing software for Internet of Things devices easier. Our analysis consists in measuring if the proposed templates are consistent with the changes performed in practical scenarios. Therefore, we automatically analyze how many commits contains evolution scenarios categorized as templates. As expected, not all commits are classified as templates, thus we also investigate the changes occurred on the remaining commits, categorizing them according to the modifications in each SPL space: FM, CK, and AM, to improve our knowledge about SPL evolution. We define some tags and manually classify these remaining commits according to the changes in evolution scenarios. Moreover, we also make preliminary automatic analysis in 13,288 commits of the Linux Kernel, between the releases 3.12 and 3.13. Such as Soletta, not all commits of Linux were classified as templates (around 14%). In summary, this work provides the following contributions: (i) an empirical study to better characterize SPL evolution, measuring to which extent evolution is safe compared to partially safe; (ii) a methodology for manually analyzing changes that might reveal novel templates for expressing evolution scenarios.

The remainder of this work is organized as follows:

- Chapter 2 presents the main concepts about SPL and its structure, aiming to support the understanding about our study.
- Chapter 3 explains how we evaluate the SPL evolution and our methodology trough our empirical study with Soletta.
- Chapter 4 contains the evaluation of our results, research questions and threats to validity.
- Chapter 5 shows the conclusion of our work. Finally, we present our contributions, related studies and future work.

2 BACKGROUND

In this chapter, we present an overview of some important concepts to improve the understanding about our work. In Section 2.1, we introduce SPL and its structure. Moreover, Section 2.2 present a overview of product line evolution, and details safe and partially safe evolution theories.

2.1 SOFTWARE PRODUCT LINE

In general, creating a single product from scratch is often massive and expensive. Mass production is a way to produce products in large scale through reuse. This approach improves productivity, reducing costs and time-to-market, however, it does not create personalized products. So, taking into account that different customers have distinct needs and wishes, it is important to increase the portfolio of products. Reuse is still used, but there are extra parts combined with the core to generate personalized products according to customers' needs. This is the basic intuition behind the idea of a production line: a set of products from a same portfolio that share a core of characteristics, and are created from reusable parts (CLEMENTS; NORTHROP, 2002).

Considering the software context, the software product line approach arises. It consists of a family of software products that share common and distinct assets providing a systematic way to generate similar products (POHL; BöCKLE; LINDEN, 2005). So, instead of generating each product from scratch, they should be developed from reusable parts. Therefore, the SPL strategy yields gains in productivity, quality, and time-to-market (CLEMENTS; NORTHROP, 2002).

A Software Product Line is usually organized into 3 high-level spaces (BORBA; TEIXEIRA; GHEYI, 2012; PASSOS et al., 2016), which we refer to as *Feature Model* (Section 2.1.1), *Asset Mapping* (Section 2.1.2), and *Configuration Knowledge* (Section 2.1.3). Each of these spaces plays a key role to generate products in a systematic way.

2.1.1 Feature Model

In SPL, the concept of each characteristic which is reused to generate products is known as *feature*. Features are used to specify and communicate the commonalities and differences of the products (APEL et al., 2016). According Feature-Oriented Domain Analysis (FODA), these features are organized into a *Feature Model* that establishes common and variable features, allowing variability management (KANG et al., 1990). The FM is usually displayed as a tree and establishes which products can be derived through hierarchy, dependencies, and constraints.

Figure 2 shows the typical diagram notation of FMs. Each feature can be *optional* (empty circle), *mandatory* (filled circle), or (filled triangle), and *alternative* (empty triangle).

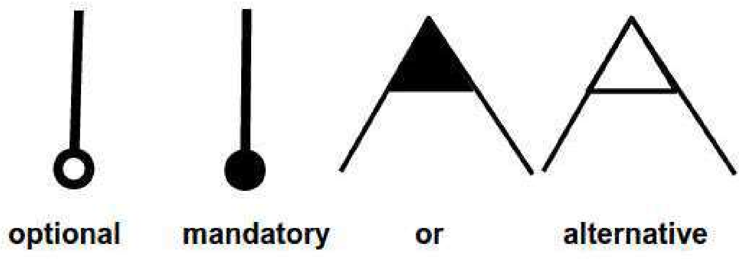


Figure 2 – Feature Notations

Optional: features which can be present or not in the SPL products;

Mandatory: features which are present in all SPL products that contain its parent;

Or: features which can be present alone or with its siblings in SPL products;

Alternative: features which are can be present in SPL products, without the simultaneous presence of its sibling (exclusive or).

Each SPL product is described as a set of features selected according to the constraints and dependencies in the FM. Figure 3 illustrates an example of FM based on features names from Soletta. The model shows these features, and its dependencies and constraints.

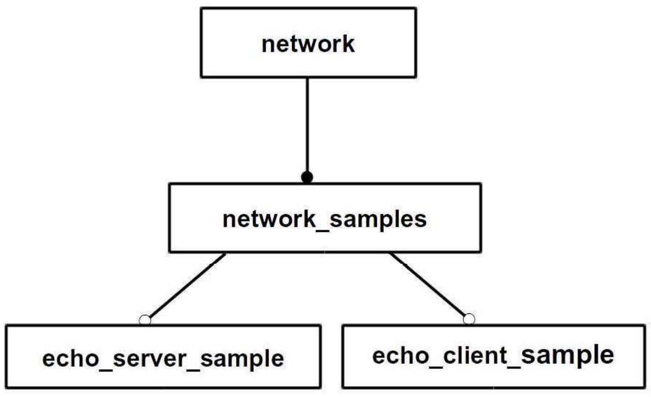


Figure 3 – Example of a Feature Model

According to the example in Figure 3, it is possible to generate the set of valid products:

1 [NETWORK, NETWORK_SAMPLES]

2 [NETWORK, NETWORK_SAMPLES, ECHO_SERVER_SAMPLE]

3 [NETWORK, NETWORK_SAMPLES, ECHO_CLIENT_SAMPLE]

4 [NETWORK, NETWORK_SAMPLES, ECHO_SERVER_SAMPLE, ECHO_CLIENT_SAMPLE]

Mandatory features should be present in all valid products, where their parent is also present. Thus, in the previous example, the NETWORK_SAMPLES feature (filed circle) is present in all valid configurations, since its parent is NETWORK. On the other hand, optional features could be present or absent in valid products, such as ECHO_SERVER_SAMPLE feature (empty circle), which is only present in some products.

2.1.1.1 Kconfig

It is important to provide a way to declare features and dependencies. Kconfig is a common tool and language used to express variability in SPL, used in projects, such as the Linux Kernel (KERNEL, 2018 (accessed August, 2018)). A FM could be spread in many different Kconfig files depending on its size, aiming to improve the organization of features. Moreover, Kconfig has been studied largely (SHE et al., 2010). The code in Listing 2.1 expresses an example of feature declaration based on the model expressed in Figure 3 captured from Soletta.

Listing 2.1 – Feature declaration in Kconfig

```
1 config ECHO_SERVER_SAMPLE
      bool "Echo server"
3     depends on NETWORK_SAMPLE && NETWORK
      select HTTP
5     default y
```

The *config* keyword (line 1) followed by a name starts a new feature configuration. Other arguments follow: A feature can be classified into types, such as *tristate* or *string*. A *depends on* (line 3) clause expresses the feature dependencies which must be satisfied so the feature can be selected. The *select* clause enforces the selection of others configs. Finally, *default* establishes the default value for the feature. Moreover, there are other keywords used in the language, such as *menu*, which could be classified as abstract features.

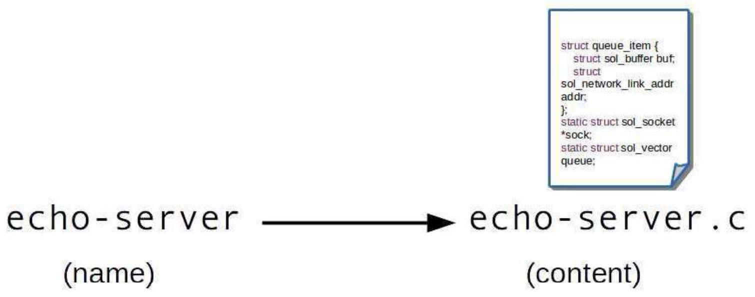
2.1.2 Assets - Asset Mapping

In Software Product Line, beyond models and features, there are also assets, usually related to implementation. Assets make these features concrete by implementing them, and can be of various forms, such as source code, documentation, or even images. Listing 2.2 shows the code associated with the `echo-server.c` file which we get from the Soletta repository.

Listing 2.2 – echo-server.c file code example

```
 1   /**
     * @file
 3   * @brief Basic echo server
     */
 5
     #include <errno.h>
 7   #include <getopt.h>
     #include <limits.h>
 9
    ...
11  #ifdef SAMPLES
     struct queue_item {
13       struct sol_buffer buf;
         struct sol_network_link_addr addr;
15   };
    #endif
17
     static struct sol_socket *sock;
19   static struct sol_vector queue;
    ...
```

Thus, the Asset Mapping (AM) consists of a mapping of asset names to the actual assets that might be used in the SPL. The representation of an asset mapping can be seen in Figure 4, which ECHO-SERVER asset name is referring into the ECHO-SERVER.C source code. However, there are other ways to associate features into source code. Listing 2.2 presents an *#ifdef* macro (line 11), which expresses that if the feature SAMPLES is selected, the code between lines 11 and 16 will be included in the product.

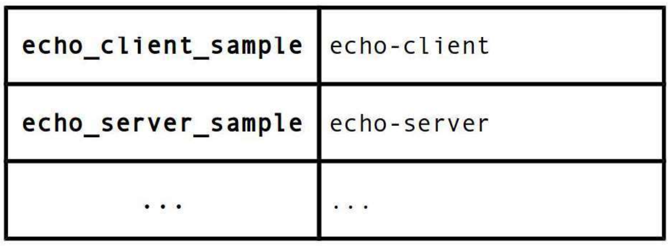


```
struct queue_item {
  struct sol_buffer buf;
  struct
sol_network_link_addr
addr;
};
static struct sol_socket
*sock;
static struct sol_vector
queue;
```

Figure 4 – Example - Asset Mapping

2.1.3 Configuration Knowledge

As aforementioned, Feature Model deals with features and its constraints and dependencies. Assets implement these features. However, we need to associate features with assets. We refer to this mapping as the *Configuration Knowledge* (CK). The CK plays a role on mapping features to their implementation, and the representation of this model is shown in Figure 5.



echo_client_sample	echo-client
echo_server_sample	echo-server
...	...

Figure 5 – Example - Configuration Knowledge

2.1.3.1 Makefile

As with the FM, there are different ways to specify the CK. In Kconfig-based systems, these are usually implemented as *makefiles*, configuration files written using the `make` language.[1] Makefiles are used in many projects, such as Linux Kernel, and also in Soletta. Listing 2.3 shows the mapping of the `ECHO_SERVER_SAMPLE` feature to the `echo-server.c` asset name, which in turn, refers to the actual asset.

Listing 2.3 – Mapping feature to asset in Makefile

```
sample-$(ECHO_SERVER_SAMPLE) += echo-server
sample-echo-server-$(ECHO_SERVER_SAMPLE) := echo-server.c
...
```

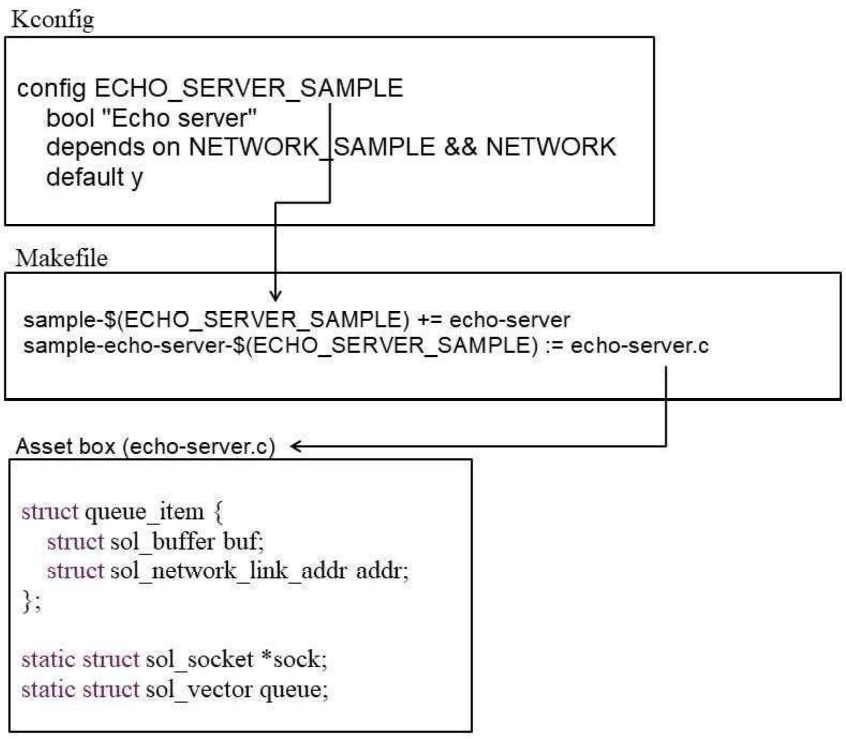
Apart from mapping features and assets, Makefile also could contains rules to specify how to build the systems, for instance, defining compilation rules and flags. Listing 2.4 shows a Makefile snippet extracted from Soletta project which contains some build rules. For instance, in line 3, the path `$(top_srcdir)src/modules/flow/` is a `flow-dir` var value. Therefore, changes in makefiles could vary further than mapping between features and assets.

Listing 2.4 – Build rules in Makefile

```
1
  # vars to distinguish the modules types
3 flow-dir := $(top_srcdir)src/modules/flow/
  linux-micro-dir := $(top_srcdir)src/modules/linux-micro/
5 pin-mux-dir := $(top_srcdir)src/modules/pin-mux/
```

Figure 6 presents the three spaces of Soletta and how they are related. In Kconfig box is shown the `ECHO_SERVER_SAMPLE` feature declaration followed by its constraints. The Asset box represents the `echo-server.c` asset. And finally, the Makefile box shows the mapping between `ECHO_SERVER_SAMPLE` feature name and the `echo-server.c` asset name.

[1] <http://www.gnu.org/software/make/manual/make.html>



```
config ECHO_SERVER_SAMPLE
    bool "Echo server"
    depends on NETWORK_SAMPLE && NETWORK
    default y
```

```
sample-$(ECHO_SERVER_SAMPLE) += echo-server
sample-echo-server-$(ECHO_SERVER_SAMPLE) := echo-server.c
```

```
struct queue_item {
    struct sol_buffer buf;
    struct sol_network_link_addr addr;
};

static struct sol_socket *sock;
static struct sol_vector queue;
```

Figure 6 – Example of the 3 spaces from Soletta

2.2 PRODUCT LINE EVOLUTION

SPLs evolve over time. Since features might be spread throughout many products, it is reasonable to affirm that modifications in SPLs could be error-prone, since a simple change might impact several products. Thus, research effort has focused on understanding SPL evolution aiming to help developers minimize the impact yielded during changes (LOTUFO et al., 2010; PASSOS; CZARNECKI; WASOWSKI, 2012; HEIDER et al., 2012).

2.2.1 Safe Evolution

The concept of *safe evolution* (BORBA; TEIXEIRA; GHEYI, 2012; NEVES et al., 2015a) aims to support developers on performing behavior-preserving changes. That is, all existing products should maintain their observable behavior after the change. Examples of such changes include code refactorings and adding optional features without changing existing code.

If we suppose an evolution scenario that performs an addition of an optional feature which is represented in Figure 7. It contains 2 sides, left and right, represented by L and L', in which L refers to the initial SPL (before the change), and L' referring to the resulting SPL (after the change). This change can be considered safe if and only if the behaviour of the existing set of valid products is preserved. The $\sqsubseteq$ symbol represents the *refinement* notation, therefore, L' refines L (in other words, L is refined by L'). Figure 8 shows that the new set of valid products from L' maintain the same behaviour of the previous products from the initial SPL L.

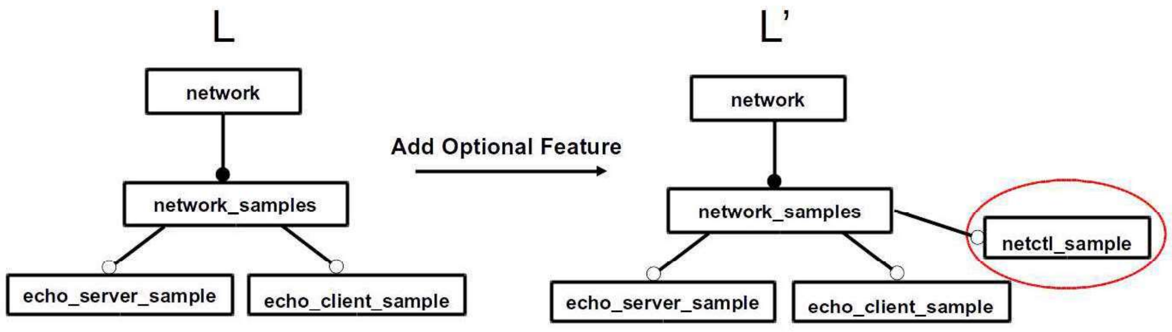


Figure 7 – Example - Evolving a SPL adding a new optional feature

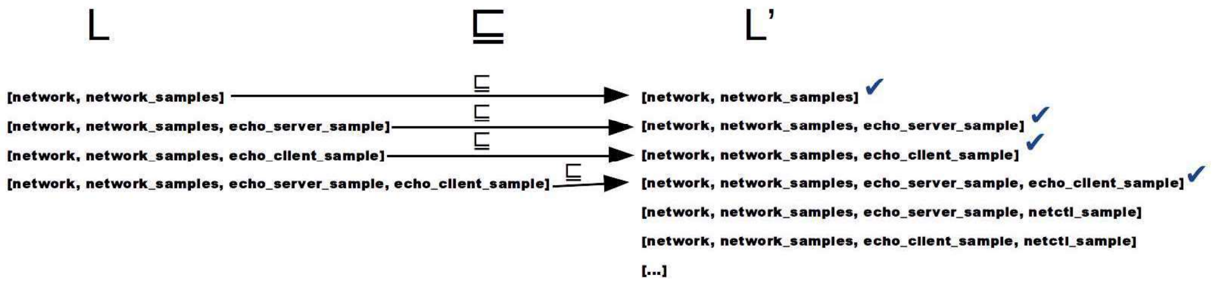

L
⊑
L'
[network, network_samples]
[network, network_samples, echo_server_sample]
[network, network_samples, echo_client_sample]
[network, network_samples, echo_server_sample, echo_client_sample]
⊑
⊑
⊑
⊑
[network, network_samples] ✓
[network, network_samples, echo_server_sample] ✓
[network, network_samples, echo_client_sample] ✓
[network, network_samples, echo_server_sample, echo_client_sample] ✓
[network, network_samples, echo_server_sample, netctl_sample]
[network, network_samples, echo_client_sample, netctl_sample]
[...]

Figure 8 – Example - Products refined after add a new optional feature

A refinement theory formalizes this concept (BORBA; TEIXEIRA; GHEYI, 2012), allowing the derivation of transformation templates (NEVES et al., 2015a; TEIXEIRA et al., 2015; BENBASSAT; BORBA; TEIXEIRA, 2016). These templates abstract common changes, capturing properties of the initial and evolved SPLs so developers only need to reason over templates, instead of the formal definitions. For instance, Figure 9 shows the ADD NEW OPTIONAL FEATURE template, which depicts adding an optional feature to an SPL. A template has a left-hand side (LHS) pattern and a right-hand side (RHS) pattern, establishing syntactic and semantic conditions for applying a transformation. We use meta-variables to represent SPL elements. If the same meta-variable appears in both sides, the element is unchanged. Therefore, we see that we add a new optional feature O, together with its corresponding asset (a') and a new mapping from an arbitrary formula e' to the new asset.

Besides syntactic conditions, we also need to fulfill semantic conditions expressed in the lower part. For this template, we can use any arbitrary expression e', provided that it is true when O is selected in a product. Both the O feature and the asset name n' must be new elements. Finally, the new products resulting from adding O must be well-formed. According to the template, after adding the new feature, all existing products are refined, since we only introduce new products, and do not change the existing ones.

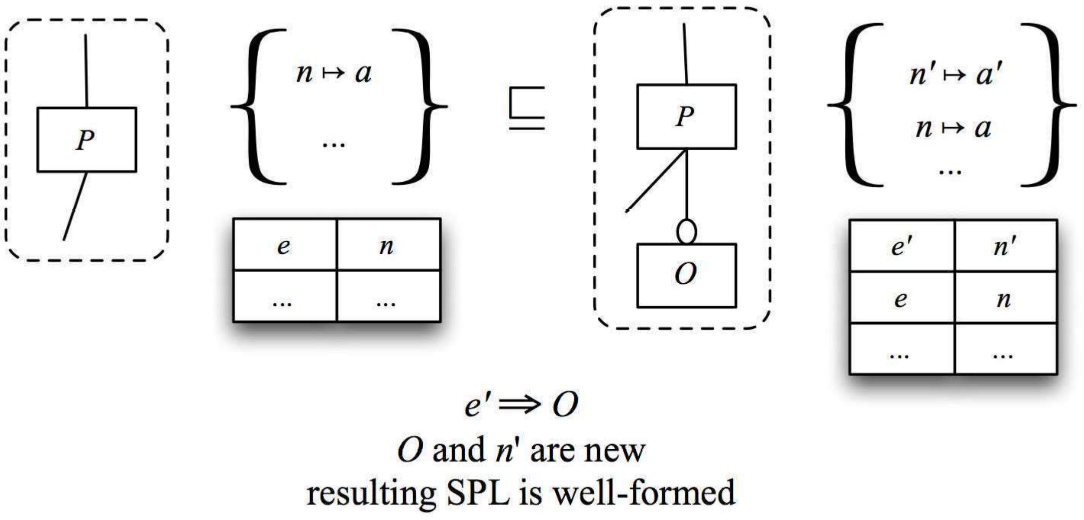


Figure 9 – Add New Optional Feature template.

2.2.2 Partially Safe Evolution

During the SPL evolution history, some changes are not consistent with the safe evolution notion. Many useful changes do intend to change the behavior of at least some of the existing products, such as bug fixes, or removing features. To provide support for developers on these types of evolution scenarios, the concept of *partially safe evolution* was proposed (SAMPAIO; BORBA; TEIXEIRA, 2016), formalized through an extension of the SPL refinement theory. The intuition for this notion is that even though a change might not preserve the behavior of all products, it might preserve the behavior of a subset of the existing products. In the extreme scenario, a change might affect the behavior of all products, and thus there is no support provided by the theories, might be c. However, this scenario might be classified as unsafe.

Figure 10 shows a partially safe evolution scenario example, which consists in remove the *netctl_sample* feature. The LHS of the figure 11 exhibits the valid products from the previous SPL, which contained the *netctl_sample* feature, and the RHS shows the valid products after the feature removal. In fact, not all previous valid products from the initial PL are refined after the change, thus, only a subset of the products are refined. In other words, this evolution scenario partially refined the SPL. More specifically, in this case, all previous products do not contain the removed feature have their behavior preserved.

Sampaio et al. (SAMPAIO; BORBA; TEIXEIRA, 2016) also suggested partially safe templates from some common scenarios which impact existing SPL products. These templates

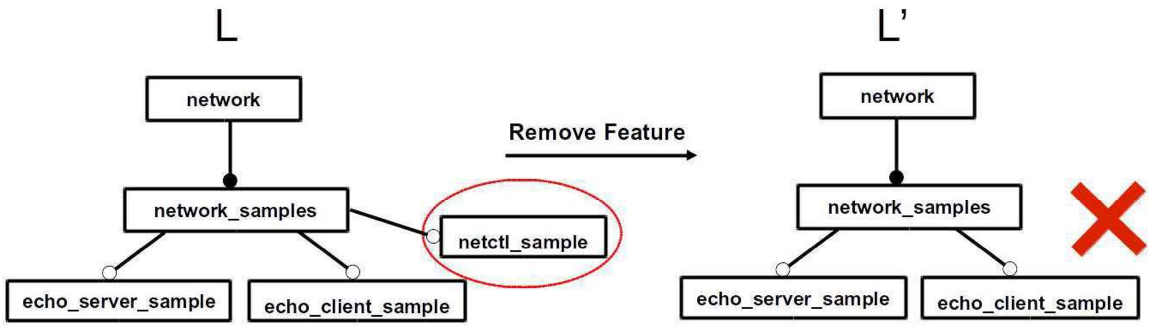


Figure 10 – Example - Evolving a SPL removing some feature

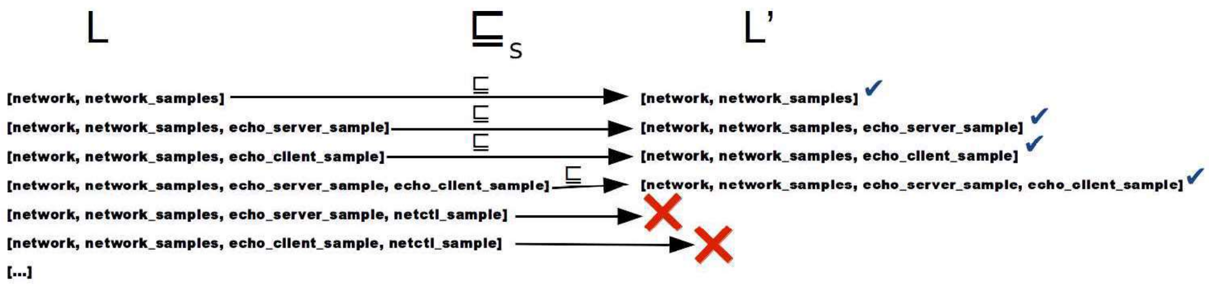

L
$\sqsubseteq_S$
L'
[network, network_samples]
[network, network_samples, echo_server_sample]
[network, network_samples, echo_client_sample]
[network, network_samples, echo_server_sample, echo_client_sample]
[network, network_samples, echo_server_sample, netctl_sample]
[network, network_samples, echo_client_sample, netctl_sample]
[...]
$\sqsubseteq$
[network, network_samples] ✓
[network, network_samples, echo_server_sample] ✓
[network, network_samples, echo_client_sample] ✓
[network, network_samples, echo_server_sample, echo_client_sample] ✓

Figure 11 – Example - Products partially refined after remove feature

also establish some requisites to precisely define the products that have their behavior preserved. For instance, Figure 12 represents the Remove Feature template, which abstracts the previous partially safe evolution scenario shown in Figure 10. By following the established syntactic and semantic rules, refinement holds for a specified subset of products S. We observe that we remove the O feature from the initial FM (F), resulting in F'. We also remove mappings referencing O from the CK. We also remove assets associated with O from the AM.

There are also semantic conditions, such as ensuring that the expressions in the mapping are related to O ($e' \Rightarrow O$), and that no other mappings refer to O in the CK. The template also defines the set of products (S) that have the behavior preserved after the change. We use the operator to establish that any valid configuration from F that does not include O has its behavior preserved. Finally, we also need a well-formedness condition. Since we assume that assets are removed, we cannot guarantee that existing products remain well-formed, except those in S.

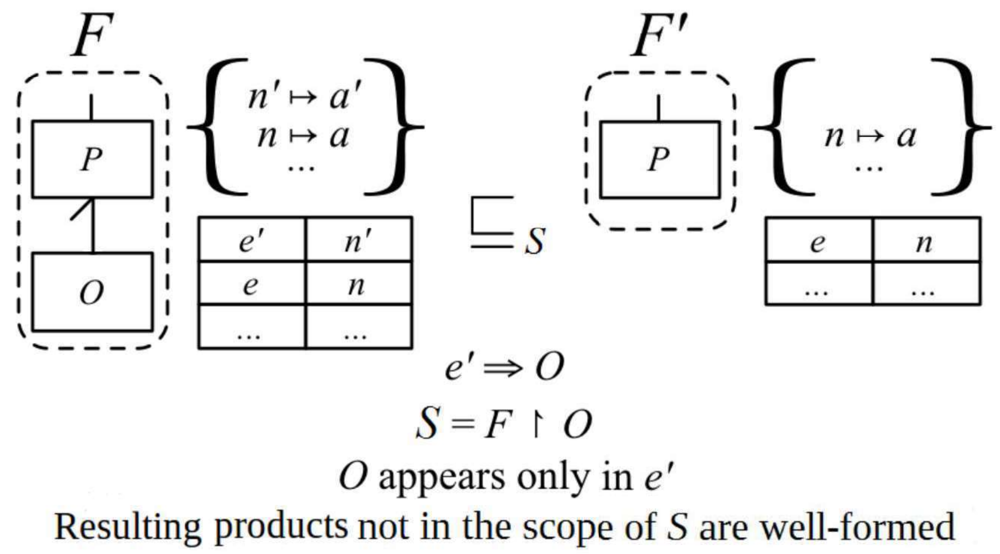


Figure 12 – Remove Feature template.

3 EMPIRICAL STUDY OF SPL EVOLUTION SCENARIOS

This chapter details the setup of our Empirical Study, aiming to categorize SPL evolution scenarios into safe or partially safe. Therefore, this chapter presents the research questions which guide our study (Section 3.1), a description of the project used as object of our analysis (Section 3.2), and further describes our methodology (Section 3.3).

3.1 RESEARCH QUESTIONS

Likewise single software systems, Software Product Lines often need to evolve. Improve code or add new functionality are common changes during products development. Adding a new feature could increase exponentially the amount of valid configurations in SPL, and a single feature is often spread in several products. Therefore, making changes in SPL is error-prone, because a simple change can impact a range of valid products. Hence, this context motivate researchers studies SPL evolution. Some works suggested patterns evolution and formalizes templates aiming understanding and support development. However, existing works that evaluate templates focus only on evaluating occurrences of either safe or partially safe evolution, but not both. Therefore, we believe it is important to collect empirical evidence over how such scenarios happen, so we can better understand how to support developers. So, these issues motivate us to perform our study. To guide our research, we use the GQM approach, such as follow (BASILI, 1992).

Goal: Our goal is to characterize SPL evolution as a whole, measuring to what extent the evolution history is safe compared to partially safe. To guide our study, we established the following research questions:

- **RQ1:** How are changes distributed in terms of safe and partially safe during the history of a software product line?
- **RQ2:** How often templates cover these real scenarios?

Metrics: To answer **RQ1**, we mined 2,300 commits from an existing SPL (each commit is considered as an evolution scenario) and classified each one as *safe* or *partially safe*, observing the distribution of evolution type over time. To answer **RQ2**, we automatically measure the occurrence of nine templates from the existing template catalogue (GROUP, 2018 (accessed november, 2018)). For the remaining commits, first we automatically divide changes according to the modified spaces (FM, AM, and CK), and then we manually characterize them using *tags*, whose frequency might reveal potential novel templates.

3.2 SAMPLE

As our object of study, we mine Soletta, a development framework with the goal of easing software development for IoT devices. Its GitHub repository currently contains 3,086 commits. We choose this project since it is structured similarly to Linux (a project studied largely), using Kconfig to manage variability, C as the main programming language, and Makefiles to map features into their implementation. Soletta is also smaller than Linux, which makes it amenable to manual analysis. Moreover, we investigate changes from the beginning of the project into the first release, aiming to understand the evolution scenarios during this lifecycle, since its conception into the first version. Thus, we analyze 2,300 commits from the evolution history, ranging from June/2015 to April/2016.

Furthermore, we have conducted a preliminary study with commits from the Linux Kernel through our automatic analysis. Then, we analyze an available graph dataset with 13,288 commits between releases 3.12 and 3.13, ranging from February/2013 to December/2013.

3.3 METHODOLOGY

Figure 13 presents the methodology we used in our study. In the top side, we illustrate how we extracted information from the repository, while in the bottom side we show how we analyzed such data. We used three tools (see Steps 1.2, 1.3, and 2.5) in our evaluation: Feature EVolution ExtractoR (FEVER) (DINTZNER; DEURSEN; PINZGER, 2016), Neo4j,[1] and Repodriller.[2]

FEVER (DINTZNER; DEURSEN; PINZGER, 2016) is a tool for mining git repositories in a feature-oriented way. Although FEVER has been developed based on the Linux structure, Soletta is a project that follows a similar structure, which makes the tool to work as expected (Step 1.1). So, we use FEVER to mine Soletta and Linux (Step 1.2) and extract certain information from commits, resulting in an output stored in a neo4j dataset (Step 1.3). This graph dataset contains nodes (*entities*), edges (*relations*), and properties (*attributes*) for each commit extracted. Changes to the Feature Model, Configuration Knowledge, and Asset Mapping are defined as `FeatureEdit`, `MappingEdit`, and `SourceEdit` entities, respectively. In FEVER, the three SPL spaces can also be expressed as `ArtefactEdit` entity that contains an attribute *type* which range among: `vm`, `build` or `source`.

Figure 14 shows the nodes, relations, and properties of the FEVER dataset that we used to derive our queries. Commit entity contains some properties (`hash`, for example), and could be connected with some entities, such as `FeatureEdit`, `MappingEdit`, and `ArtefactEdit`.

[1] <https://neo4j.com>
[2] <https://github.com/mauricioaniche/repodriller>

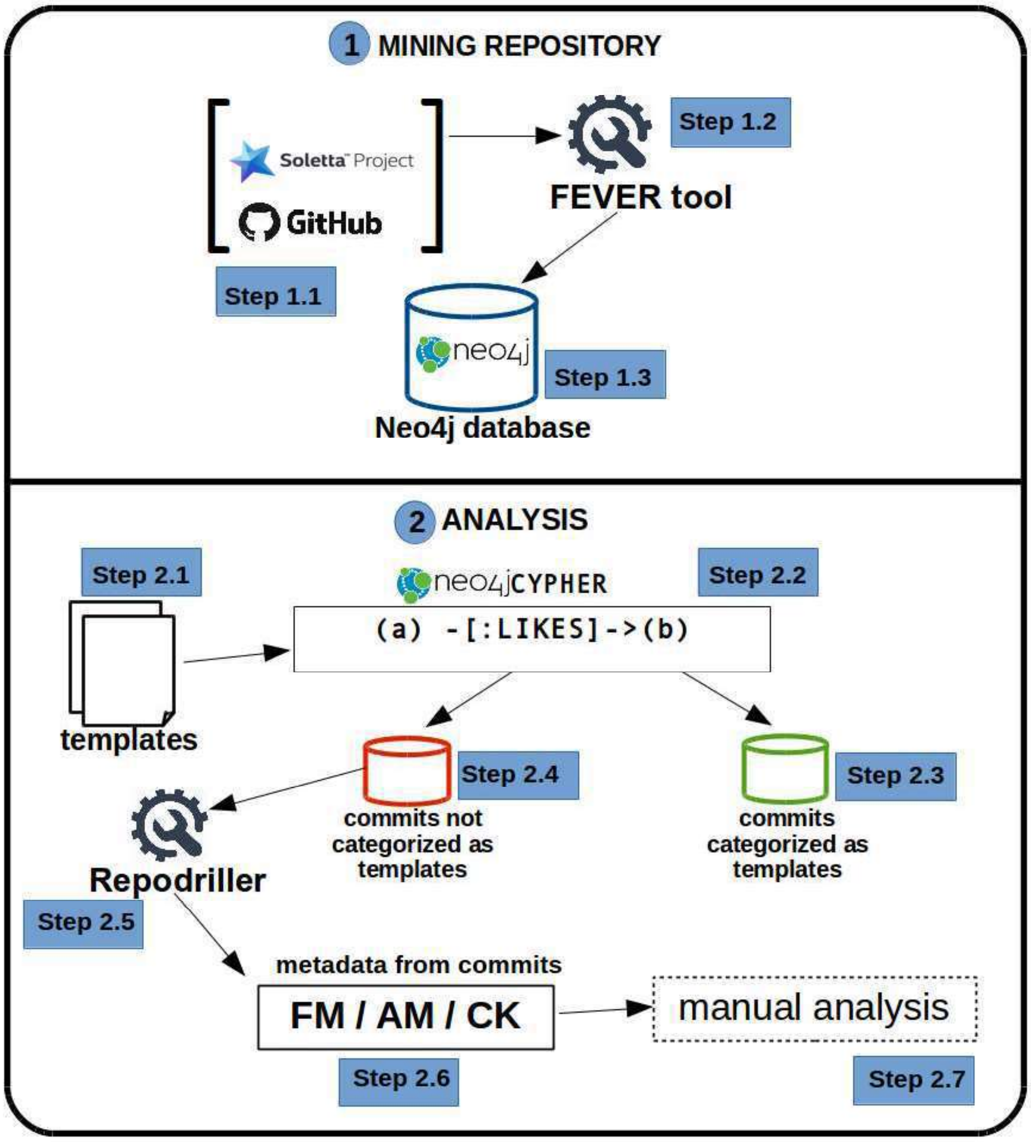


Figure 13 – Overview of the design study

Listing 3.1 – example of query which captures commits that changes VM

```
match (c:commit)-->(a:ArtefactEdit)
where a.type = "vm"
return distinct c, a
```

To access this data we use the neo4j platform and perform queries in the *cypher* language (Step 2.2). For illustrate an query example, suppose some way to capture all commits which modify the Feature Model. For that, the query should specify that `commit` entity must have be connected with some `ArtefactEdit` entity with type value *vm* (see listing 3.1).

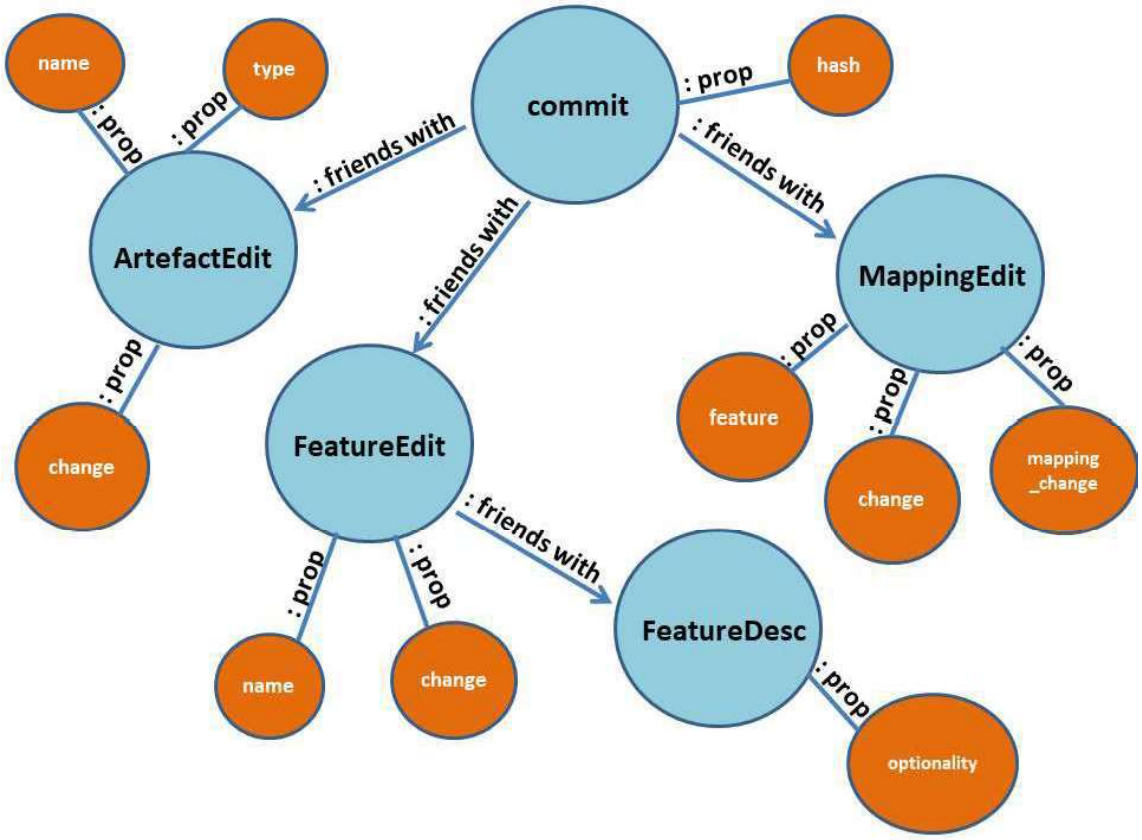


Figure 14 – Graph representation of nodes, relations, and properties in FEVER dataset

3.3.1 Capturing commits categorized by Templates

As mentioned in Chapter 2, there are safe and partially safe evolution notions which characterize evolution scenarios in SPLs. Previous works based on these theories suggested templates to avoid errors and supports development (NEVES et al., 2015b; SAMPAIO; BORBA; TEIXEIRA, 2016; BENBASSAT; BORBA; TEIXEIRA, 2016). Aiming to investigate how often these templates occurs in practical scenarios, we use queries to encode nine existing templates (Step 2.2). Table 1 shows these nine templates that we choose (four safe and five partially safe) to make our analysis (Step 2.1). We express the safe templates in queries created from scratch according its constraints and specification. On the other hand, for partially safe templates we reuse existent queries based on previous work (SAMPAIO; BORBA; TEIXEIRA, 2016).

All templates used through our research are available on Appendix A. We describe each one of them below:

Add new Optional Feature This template expresses an evolution scenario in which is added a new optional feature, and a new asset which implements this feature, and also a new association between the feature and the asset. So, the evolution should change: FM, adding a new feature in Kconfig file; CK: adding a new mapping in Makefile file; AM: adding a new asset related to the mapped file added in Makefile.

Safe Templates	**Partially Safe Templates**
Add new optional feature	Change asset
Add any feature without change CK and AM	Remove feature
Remove unused assets	Change CK lines
Add unused assets	Add assets
	Remove assets

Table 1 – Partially Safe and Safe Templates used in our study

Add any Feature without change AM and CK (safe) This templates represents a change which only added some feature, modifying the FM without changing the AM or CK. This kind of change occurs to add some abstract feature, aiming to improve the variability model.

Remove unused assets (safe) This template represents removal assets scenarios, however, these assets should be unused. Moreover, the change should not modify CK and FM spaces.

Add unused assets (safe) This template expresses the addition of some asset without any mapping in the CK. The change should also not modify the FM.

Change asset (partially) This template specifies a scenario of some modifying an asset. As constraint, the change should not contain added nor removed of assets. Furthermore, the change should be only in the AM, so CK and FM remain unchanged.

Remove feature (partially) To express the scenario of a removal feature it is necessary to modify the three SPL spaces. So, this change modify FM removing the feature, change CK deleting the mapping, and also remove the related asset which implements the feature removed.

Change, Add, Remove CK lines (partially) This template represents changes that affect only the CK. So, the evolution scenario should not alter FM or AM spaces.

Add assets (partially) This template characterizes an addition of a new mapping in CK and also an addition of the new related asset. Therefore, this pattern modifies the CK and AM spaces, preserving the FM.

Remove assets (partially) This template is the opposite of Add assets, thus characterizing the removal of both the existing mapping in CK and also the related asset. As Add assets template, Remove assets modifies the CK and AM spaces, preserving the FM.

Templates	FM	CK	AM
Add new Optional feature	x	x	x
Add any feature without change CK and AM	x		
Remove unused assets			x
Add unused assets			x
Change asset			x
Remove feature	x	x	x
Change CK lines		x	
Add assets		x	x
Remove assets		x	x

Table 2 – Changed Spaces by Template

The summary of the modified spaces according to each template is shown in Table 2.

First, it is important to understand the templates to generate queries. For instance, Figure 9 shows the Add new Optional Feature template, which depicts adding an optional feature to an SPL. So, as previously explained, Add new Optional Feature represents a evolution scenario which modifies the three SPL spaces: FM, CK, and AM.

Then, after understanding the templates and the spaces affected by the changes, we can derive queries. Listing 3.2 illustrates an example of query used in our work which expresses the Add new optional feature template. So, this query returns all hash commits that present an addition of optional feature in its evolution scenario. For that, the query must have a `commit` entity connected with `FeatureEdit` entity which contains `change` property with value *Add* (line 2). This commit should also be connected with some `MappingEdit` entity which contains `feature` property value equal to the `FeatureEdit` name property (line 4), to guarantee that the feature and the mapping are related. Further, the commit must be connected with some `ArtefactEdit` with `source` type value and `add` (line 1) change value properties. This commit should not modify nor remove some asset (lines 6 and 7). All these constraints returns the commits which contain changes in FM, CK and AM in a way that categorize an Add new Optional Feature template.

Listing 3.2 – query - Add new optional Feature

```
1 match (ae:ArtefactEdit {change: "ADDED"})<--(c:commit)
  -[]->(f:FeatureEdit {change:"Add"})-[]->(fd:FeatureDesc {optionality:"optional"})
3 WHERE
  (c)-->(:MappingEdit {feature:f.name}) and
5 (c)-->(:ArtefactEdit {type: "source" , change: "ADDED"}) and not
  (c)-->(:ArtefactEdit {type: "source" , change: "MODIFIED"}) and not
7 (c)-->(:ArtefactEdit {type: "source" , change: "REMOVED"})
  return distinct c.hash
```

Then, we repeat this process to express templates in queries for each selected safe evolution templates. The queries used to capture the partially safe scenarios were based on previous works (SAMPAIO; BORBA; TEIXEIRA, 2016), as we mentioned before. Change

Asset is one of the partially safe templates used in our study, and Listing 3.3 represents the query to capture scenarios that match this template.

Listing 3.3 – query - Change Asset

```
1 match (c:commit)-->(a: ArtefactEdit {change:"MODIFIED"})
  where
3 not (c)-->(:ArtefactEdit{change:"ADDED"}) and
  not (c)-->(:ArtefactEdit{change:"REMOVED"})and
5 not (c)-->(:ArtefactEdit{type:"vm"})and
  not (c)-->(:ArtefactEdit{type:"build"})
7 return distinct c.hash
```

This query returns all commits which only modify an `ArtefactEdit` entity (line 1, with `change` value as MODIFIED). Furthermore, the commits should not affect some `ArtefactEdit` with type property with `vm` (FM) nor `build` (CK) values (lines 5 and 6). In other words, this query only returns changes related to `source` (AM).

In fact, not all available templates can be expressed in queries and capture precisely the scenarios. For instance, suppose hypothetically some query to capture Refine Asset template (showed in Figure 15). According to the template, one evolution scenario must modify some asset in a way that preserves the behavior. So, the changed asset should be refined (BORBA; TEIXEIRA; GHEYI, 2012). Then, to express this pattern in query, it is necessary to specify that some commit affects the AM space *modifying* some asset, without changing FM and CK. Therefore, the query would be the same as Change Asset template. Although the query returns commits which contain modified assets, there is no guarantee that those changes refine them. All queries used through our research are available on Appendix C.

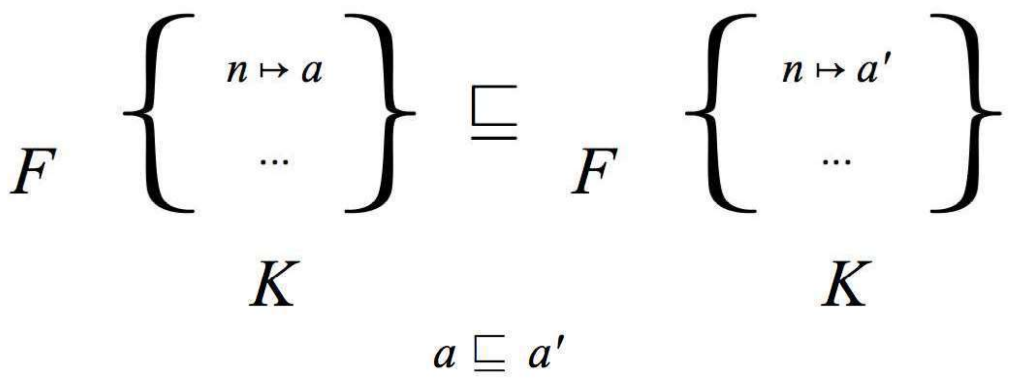


Figure 15 – Template **not** expressed in query - Refine Asset

After these steps, we collect the commits which were classified in templates. To check that queries were correctly created, avoiding wrong results, we manually checked the results for all queries, to assess their accuracy. Furthermore, aiming to mitigate the bias of only the first author developing and confirming the precision of the queries, another researcher also blindly reviewed the results of each query, checking if the commits were correctly classified by the query. After both authors reviewed all of the classified commits, the classifications were merged and revised. In cases of doubt or disagreement, a third researcher acted to reach consensus.

3.3.2 Analysis of commits not captured by Templates

After identifying the commits captured by queries, we make an approach to analyze the remaining commits not covered by templates. First of all, we divide the commits in groups according the changes in the three spaces of a SPL, and their arrangement: [AM], [CK], [FM], [FM, CK], [FM, AM], [CK, AM], [FM, CK, AM]. For each combination we use scripts in Repodriller (Step 2.5) to automatically mine the commits aiming to obtain certain information, such as path name, file name, date, amount of files added, lines modified, lines removed, and so on. The output of the mined commits was stored in a CSV file.

The list below represents the fields of our dataset which were manually filled (Step 2.7).

- **change_fm:** the type of change performed in Kconfig file;
- **Kconfig_tags:** tags to categorize the changes in Kconfig (see table 4);
- **change_ck:** the type of change performed in Makefile file;
- **Makefile_tags:** tags to categorize the changes in Makefile (see table 4);
- **chage_am:** the type of change performed in assets;
- **assets_tags:** tags to categorize the changes in assets (see table 4);
- **evolution:** classify the evolution scenario as *safe* or *partially safe*.

The rest of the data was captured automatically through scripts, as we explained before (Steps 2.5 and 2.6). We categorize each commit with *tags* according to the changes for each space: AM, CK and FM (representing assets, Makefile, and Kconfig, respectively). Table 3 contains the *tags* which we defined to make our manual analysis. Furthermore, aiming to thoroughly classify the evolution scenario, we also use *change type* to represent the type of modification for each tag. The *change type* varies among:

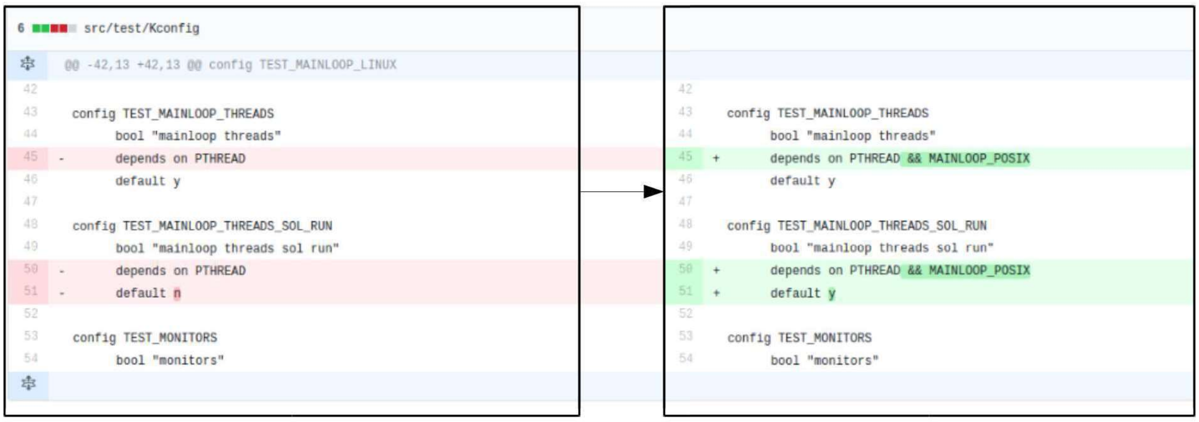
- **New:** when there is a new instance of some *tag* change;
- **Add:** if there is, at least, one instance of some *tag* change and a new is added;
- **Remove:** if there is some removal *tag* change instance;

Space	Tags	Description
AM	ifdef	if the change in AM modifies *ifdef* directive
	include	if the change in AM modifies *include* expression
	addAsset	if the change contains an addition of new asset
	changeAsset	if the change modifies some asset
	removeAsset	if the change contains removal of some asset
CK	mappingA	if the change is related to *all* mapping: LHS and RHS
	mappingL	if the change modifies only LHS (feature expression)
	mappingR	if the change modifies only RHS (assets mapped)
	ifdef	if the change in Makefile contains some *ifeq* directive
	build	if the change in Makefile contains build rules specifying how the asset files must be built
FM	feature	if the change in Kconfig modifies some *config* expression
	menu	if the change in Kconfig modifies *menu* expression
	depends	if the change in Kconfig presents some *depends on* clause
	default	if the change in Kconfig presents some *default* expression
	select	if the change in Kconfig presents some *select* expression

Table 3 – Tags to Categorize the commits not covered by templates

- **Change:** if there is some change in an already existent *tag* change;
- **Move:** when there is some instance removed in a specific local to be placed in another one;
- **Rename:** rename of some artefact (file name, variable, path).

We illustrate an example in Figure 16 to show how we tagged each evolution scenario according to changes in commits.



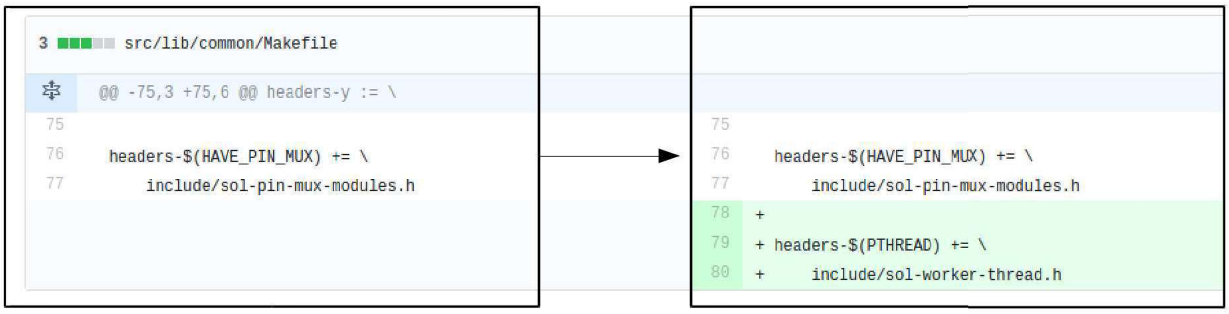
```
6 src/test/Kconfig
@@ -42,13 +42,13 @@ config TEST_MAINLOOP_LINUX

 config TEST_MAINLOOP_THREADS
       bool "mainloop threads"
-      depends on PTHREAD
       default y

 config TEST_MAINLOOP_THREADS_SOL_RUN
       bool "mainloop threads sol run"
-      depends on PTHREAD
-      default n

 config TEST_MONITORS
       bool "monitors"
```

```

 config TEST_MAINLOOP_THREADS
       bool "mainloop threads"
+      depends on PTHREAD && MAINLOOP_POSIX
       default y

 config TEST_MAINLOOP_THREADS_SOL_RUN
       bool "mainloop threads sol run"
+      depends on PTHREAD && MAINLOOP_POSIX
+      default y

 config TEST_MONITORS
       bool "monitors"
```

Figure 16 – Change in Kconfig file - hash commit: *2d39b630bf*

First, we can observe that previously (LHS) the features `TEST_MAINLOOP_THREADS` and `TEST_MAINLOOP_THREADS_SOL_RUN` already contains `depends on` clause (feature `PTHREAD` in lines 45 and 50) and `default` with n value. After the change, in the RHS, there is an **added** feature as dependency (`MAINLOOP_POSIX` in lines 45 and 50), and the default clause was changed to y value. In this way, we classify this commit as: `add` as `change_fm`, and `depends` as `Kconfig_tag`; and also `change` as `change_fm`, and `default` as `Kconfig_tag`.

We show another evolution scenario in Figure 17, which presents changes in the Makefile. According to the image, after the change (in RHS), there is a new (more one) mapping between the feature PTHREAD and `sol-worker-thread.h` asset. Therefore, we classified this evolution scenario as: `add` as `change_ck`, and `mappingA` as `Makefile_tag`.



```
3 src/lib/common/Makefile
@@ -75,3 +75,6 @@ headers-y := \

  headers-$(HAVE_PIN_MUX) += \
      include/sol-pin-mux-modules.h
```

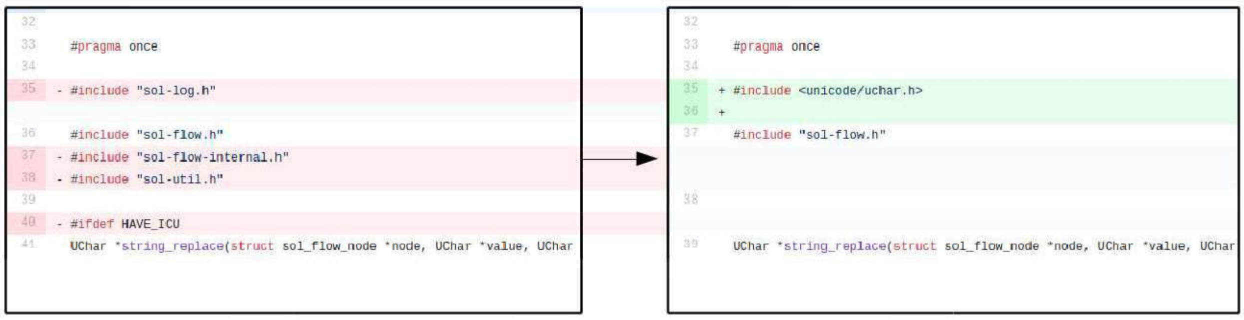
```

  headers-$(HAVE_PIN_MUX) += \
      include/sol-pin-mux-modules.h
+
+ headers-$(PTHREAD) += \
+     include/sol-worker-thread.h
```

Figure 17 – Change in Makefile file - hash commit: *4dc76a75e6*

Likewise other spaces, we also classify AM according the characteristics of each commit analyzed. Figure 18 also presents an evolution scenario (hash commit: *3c6aa6d205*) which modifies a source file. Before the change, in the LHS, there were `include` and `ifdef` directives, and after the changes, there are removed instances of `ifdef` and `include`. Hence, we classify this commit as: `remove` as change type, and `include` and `ifdef` as AM tags.

Regarding the evolution type, we adopt a conservative stance, and anytime we cannot fully guarantee that the behavior for all SPL products is preserved, we categorize the change as *partially safe* evolution. Therefore, we classified these commits (*4dc76a75e6* and *2d39b630bf*) as *partially safe*.

Similar to the manual query analysis, two researchers also make manual analysis over the remaining commits, aiming to ensure consistency over the results and avoid bias. However, in this phase there were no disagreement scenarios. All our data and scripts are available online (GOMES, 2019 (accessed february, 2019)).



```

  #pragma once

- #include "sol-log.h"

  #include "sol-flow.h"
- #include "sol-flow-internal.h"
- #include "sol-util.h"

- #ifdef HAVE_ICU
  UChar *string_replace(struct sol_flow_node *node, UChar *value, UChar
```

```

  #pragma once

+ #include <unicode/uchar.h>
+
  #include "sol-flow.h"



  UChar *string_replace(struct sol_flow_node *node, UChar *value, UChar
```

Figure 18 – Change in Asset file - hash commit: *3c6aa6d205*

4 EVALUATION

This chapter presents the results of our study in Soletta project, evaluating the commits classified as templates (Section 4.1.1) and also the commits not classified (Section 4.1.2). Further, we present our preliminary results from automatic analysis on Linux Kernel (Section 4.2). Moreover, we answer the research questions which motivates the development of this work (Section 4.3). Finally, we show the threats of validity of our study (Section 4.4).

4.1 SOLETTA RESULTS

In this section we present the results of our empirical study, evaluating the commits captured by templates (Section 4.1.1), and the remaining ones (Section 4.1.2).

4.1.1 Commits captured by templates

After encoding templates into queries to automatically classify commits, we obtained the following results. From the 2,300 commits which we analyzed, 1,810 of those were automatically classified by the queries. After analyze manually the commits captured by queries, there were doubts over 23 commits. A senior researcher analyzed those cases, and we ended up discarding 10 commits out of these 1,810, since they did not exactly match the templates: six from Add New Optional Feature, three from Remove Feature, and one from Add Any Feature without Changing CK and AM. For example, there are two instances of commits consisting of feature renaming. FEVER captures those as instances of the Remove Feature template, since the feature being renamed shows up in the commit *diff* as being removed and a supposedly new feature is added. Moreover,

Evolution	**Template**	**Commits returned by queries**	**Excluded**	**Commits classified as templates**
partially	Remove assets	0	0	0 (0%)
	Add assets	0	0	0 (0%)
	Change asset	1,662	0	1,662 (72.26%)
	Change CK lines	35	0	35 (1.52%)
	Remove feature	5	3	2 (0.09%)
safe	Add new optional feature	62	6	56 (2.44%)
	Add feature (no CK and AM)	6	1	5 (0.21%)
	Remove unused assets	6	0	6 (0.26%)
	Add unused assets	34	0	34 (1.48%)

Table 4 – Amount commits returned by queries

the ADD ANY FEATURE WITHOUT CHANGE CK AND AM scenario which we excluded (*d74314d174*) presents a moved feature (removed in any file, and replaced in other), so FEVER captured this commit as an added feature. Further, between the six commits excluded with ADD NEW OPTIONAL FEATURE classification, we removed three of those due the same reason, presenting a mapping in CK which relate the new feature added with some already existing asset. We also excluded two others commits which modify CK mapping a feature into assets, however, this feature is different than the feature added in FM. Finally, we excluded the remaining one because the asset added was mapped to an existing mapping in CK. In other hand, CHANGE ASSET, CHANGE CK LINES, REMOVE UNUSED ASSETS, and ADD UNUSED ASSETS do not present commits excluded during manual checking.

We ended up with 1,800 commits automatically classified into templates, representing 78.3% of the entire sample. Table 4 shows the amount of commits yielded for each of the templates, after performing the manual analysis of the query precision. The results presented in Table 4 show that the CHANGE ASSET template is the most frequent template throughout the history of Soletta. This template consists of arbitrary changes to assets, without modifying the FM and CK. In contrast, the queries for the partially safe templates ADD ASSETS and REMOVE ASSETS did not yield any commits. These templates represent adding (or removing) a mapping in CK, together with an added (or removed) asset. We believe this is due because it is more frequent that changes occur in assets already mapped in the Makefile rather than adding or excluding both mapping and asset files together. Such as the evolution scenario in this commit: *c065eebc42*.[1] In this case, a new asset SOL-LIB-LOADER.O was added in an already existing mapping in Makefile which refers to feature PLATFORM_LINUX.

Figure 19 represents a timeline of the number of commits per month for each template used in our work, except for CHANGE ASSET, which due to its high occurrence, would difficult the visualization of the other commits. So, according to the plot, we can observe that although ADD NEW OPTIONAL FEATURE is spread throughout the period considered in this evaluation, there are more instances of the template in the beginning of the project. There are few instances of the REMOVE UNUSED ASSETS template, and surprisingly, most of them occur on the beginning of the project. All of these commits present a commit message expressing that the deleted assets were stale. The low number of occurrences allow us check each one manually, however, this task could be automated using a text search engine library. In our sample there are only two instances of the REMOVE FEATURE template. The query actually yields five commits, but there were three false-positives which we manually excluded, such as renaming cases, as previously mentioned. We argue this low occurrences due to Soletta being a recent project, and removal features scenarios happens as they become obsolete. The ADD UNUSED ASSETS template

[1] https://github.com/solettaproject/soletta/commit/a54f22ebcc

had 34 occurrences. In fact, adding some asset without associating it with some feature preserves behavior. However, these assets might be mapped to some feature later in the evolution history. In this case, it might be the case that this later change does not preserve the behavior for some of the products. According to the plot exhibited in Figure 19, the Change CK Lines template had more instances in beginning and the end of project.

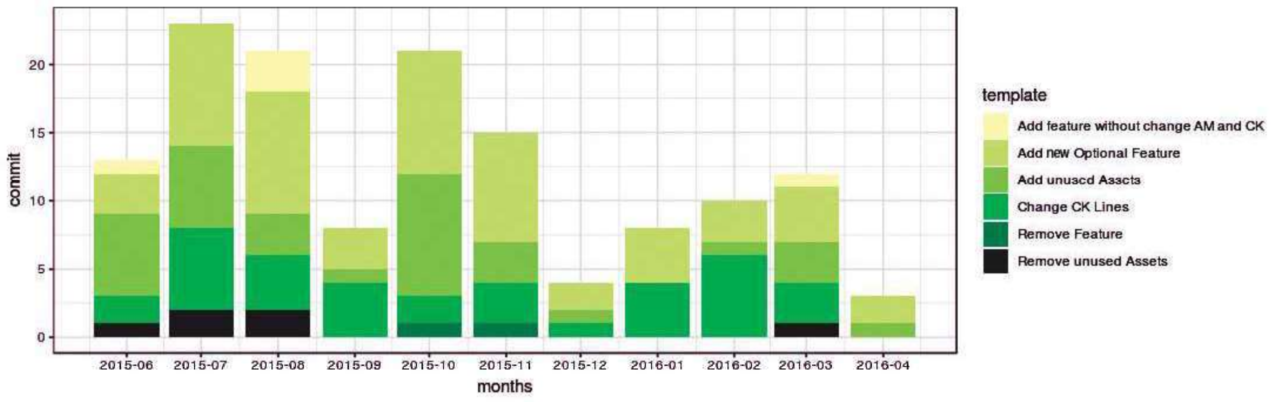


Figure 19 – Timeline Templates - *without change asset*

4.1.2 Commits not covered by templates

After classifying 1800 commits with templates, we perform analysis on the 500 remaining commits (RC) not covered. As we mention in Chapter 3, we use some scripts in Repodriller to automatically collect commits aiming to obtain certain information, such as date, lines added, lines removed, amount of files, among others, from all spaces and its combinations. Figure 20 shows a quantitative illustration from remaining commits for each space combination.

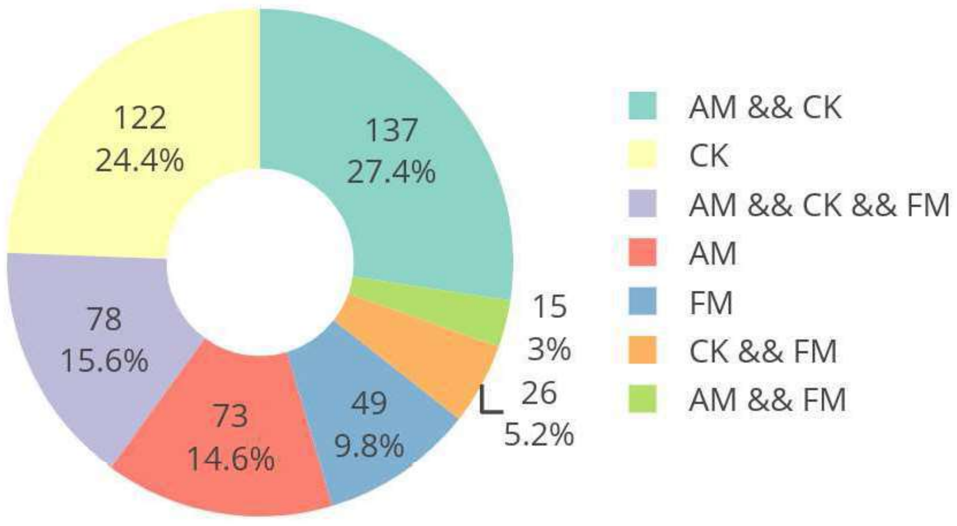


Figure 20 – Modified Spaces in Remaining Commits

This section presents our manual analysis of the remaining commits, that is, all commits that are not automatically categorized as templates. First of all, we divide the commits in groups according to changes in the SPL spaces, and their combination: [AM], [CK], [FM], [FM, CK], [FM, AM], [CK, AM], [FM, CK, AM]. Figure 20 shows a quantitative illustration from remaining commits for each space combination. We observe that the most frequent change pattern is that of modifying the CK and AM, resulting in 137 commits from an amount of 500, representing 27.4% of the remaining commits. The second most representative pattern considers changes performed solely into the CK, consisting of 24.4% of the commits. The combination with the fewest number of commits is the one that modifies both AM and FM spaces at the same time, representing 3%.

We then classify each commit according to the *change type* and *tags* specified in Chapter 3.3.2. Recall that each modified space (AM, CK, or FM) in the commit could be classified with more than one *change type* and *tags*, as we mentioned before according to Figure 16. In what follows, we present some results.

AM changes (5.5% safe vs 94.5% partially safe). Among the changes performed only in assets, 83% modify and add assets in the same commit. From all commits, only 5.5% present `ifdef` directives. On the other hand, the `include` tag is present in 35.6% of the commits. To precisely determine if a change in an asset affects the behavior of an existing product, we would have to run tests or use some verification technique. Therefore, similar to the Change Asset template, we conservatively establish that arbitrary changes to assets are *partially safe*. Therefore, there could be more instances of safe evolution scenarios than those we have classified.

CK changes (19.7% safe vs 80.3% partially safe). In evolution scenarios that only modify the Makefile, several commits present changes related to build rules (about 65%). Fewer are related to `mapping` features and assets (around 12.3%). Moreover, only 4.9% commits were classified with the `ifeq` tag. We argue that the high occurrence of `build` compared to the low number of `mapping` and `ifeq` tags are due to changes in CK being followed by changes in other SPL spaces.

FM changes (8.2% safe vs 91.8% partially safe). Most of the changes which modify the Kconfig only are related to the `depends on` clause (60.41%). From all of those instances, 32% insert the first dependency related to a feature (**new** *change type*). The `default` and *select* tags are present in 14.6% of the commits. Only three commits modify features, where two move features and one removes a feature (classified according to each respective *change type*).

AM and CK changes (21.2% safe vs 78.8% partially safe). Considering this subset of commits, only 24.8% of changes to the code (AM) are related to `ifdef` directives, and 40.8% present `include` tags. On the other hand, around 16% of CK changes present `ifdef` directives. Considering only the AM space, changes add assets in 43% of the commits, while if we consider only the CK, changes modify the mapping in 57%

of the commits. Otherwise, 36.5% of the commits perform both changes at the same time: changing the mapping and adding a new asset. As we mentioned before, there are more *mapping* instances in AM and CK group than commits which only modify CK. We also tagged 41.6% commits as changes involving *build* rules. Moreover, we found three safe evolution scenarios that are consistent with the Split Asset safe evolution template (BORBA; TEIXEIRA; GHEYI, 2012).

CK and FM changes (38.5% safe vs 61.5% partially safe). Considering the changes to the FM, we observe that 42.3% commits change feature dependencies. Most of those changes increase the number of dependencies (`add` *change type*). Analyzing changes to the CK, we observe that 65.4% of commits were classified with the *mapping* tag. In contrast with the CK group, only 19.3% from all commits present changes related to build rules. Moreover, 23% of all commits from this group involve an addition of *depends on* clause in the Kconfig together with changes to mapping in the Makefile.

AM and FM changes (33.3% safe vs 66.7% partially safe). Among the AM changes, there are no additions nor removals, only changed assets. In the FM, few commits add features (around 26%), but in contrast, 40% modify the `depends on` clause. Evaluating the AM space separately, 26.6% and 13.3% from all commits were classified, respectively, with the `include` and `ifdef` tags.

AM, CK and FM changes (23% safe vs 77% partially safe). Around 13% of the evolution scenarios that simultaneously modify the three SPL spaces contain an instance of the Add New Optional Feature template together with some other type of change in the commit, such as changes to `ifdef` directives. Among those instances there are three commits (hashes: *3556363e0f*, *7edb9d26bf*, *ecf696de10*) which only contain Add new Optional Feature template (three false-negatives), which were not captured by FEVER. We believe that this is due to the fact that certain kinds of assets from Soletta were not properly captured by the FEVER tool (for instance, .json or .fbp sources, and some documentation files). Observing each space individually, FM presents 19.2% of `depends on` changes, less than evolution scenarios which only modify the FM. In contrast, modifications over the `config` expression account for 64% of changes. Regarding the CK, 79.5% of changes are categorized with the `mapping` tag, and only 11% change build rules. Around 36% of the changes to the AM are associated to the `ifdef` tags, while 37% are related to `include` tags. Moreover, 52% of commits present an added asset, but in contrast, only two commits remove assets.

We plot one timeline for each SPL space separately. Then, Figure 21 shows a timeline with the `tags` used to classify commits which changes at least the FM in Remaining Commits ([FM], [AM and FM], [CK and FM], and [AM, CK, and FM]). Figure 22 presents a timeline with CK `tags` through groups in Remaining Commits which modify at least the Makefiles ([CK], [AM and CK], [CK and FM], and [AM, CK, and FM]). Finally, Figure 23 also shows a timeline with *tags* which classify commits modifying the AM space, in other

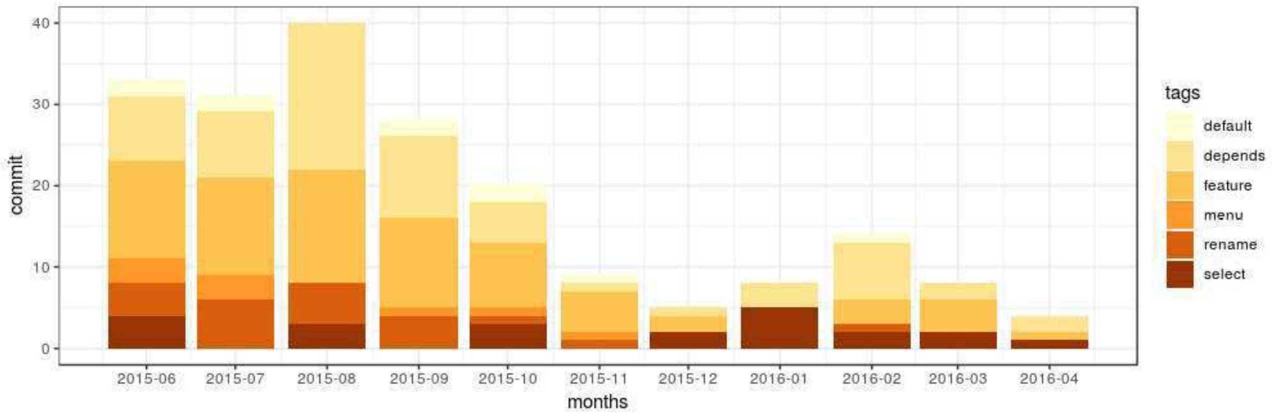


Figure 21 – Timeline of FM Tags in Remaining Commits

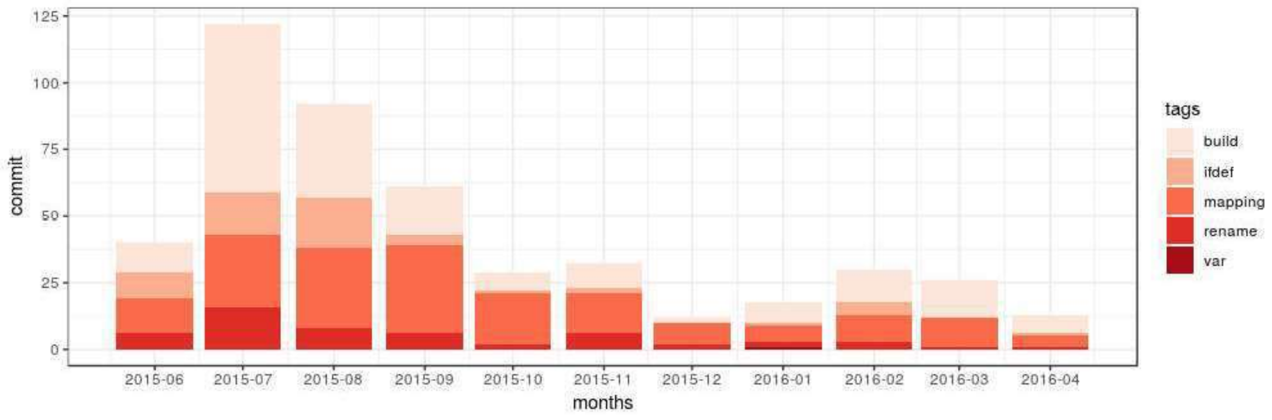


Figure 22 – Timeline of CK Tags in Remaining Commits

words, groups which modify assets ([AM], [AM and CK], [AM and FM], and [AM, CK, and FM]).

In summary, Table 5 presents the results of AM tags from all remaining commits which modify assets. As expected, most of these commits were classified with `changeAsset` tag.

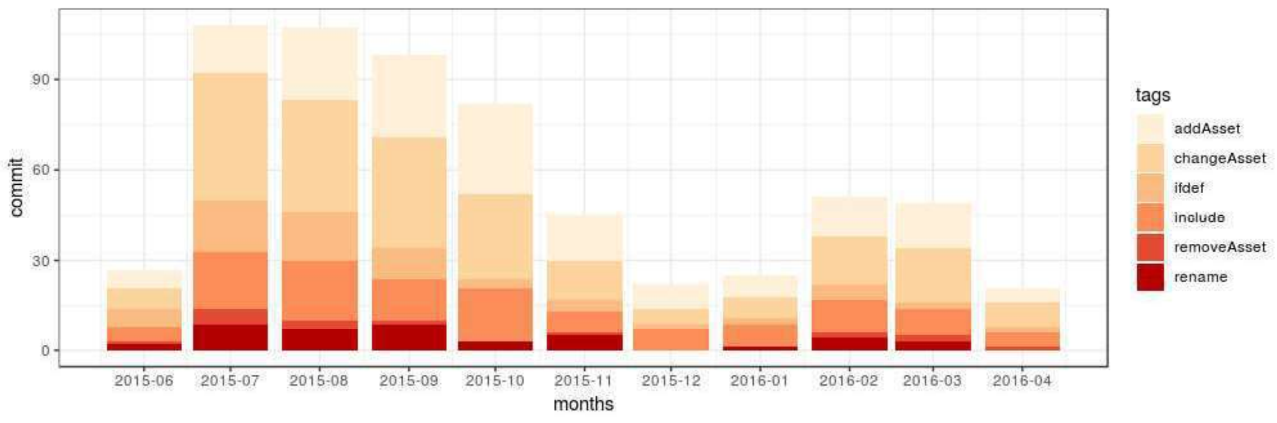


Figure 23 – Timeline of AM Tags in Remaining Commits

	AM				
	include	ifdef	changeAsset	addAsset	remAsset
GROUPS					
AM	26 (35.62%)	4 (5.48%)	70 (95.9%)	66 (90.4%)	6 (8.2%)
CK	—	—	—	—	—
FM	—	—	—	—	—
AM CK	55 (40.1%)	34 (24.8%)	89 (65%)	59 (43.1%)	8 (5.8%)
AM FM	4 (26.7%)	2 (13.4%)	10 (66.7%)	0 (0%)	0 (0%)
CK FM	—	—	—	—	—
AM CK FM	36 (46.2%)	28 (35.9%)	49 (62.8%)	41 (52.6%)	2 (2.6%)

Table 5 – Summary of tags from remaining commits which modify AM space

In the same way, several commits present `addAsset` tags. On the other hand, few commits contains changes related to `ifdef` directives. Table 6 shows the rating for each CK tag according the groups which modify makefiles. To illustrate an example in Table 6, we highlighted one value with blue and bold style, and according to this value we can assume that 65% of commits from CK group have changes related to `build` rules. The highlighted red value shows that changes which modify the three SPL spaces ([AM, CK, and FM] group) presents few changes related to `build`. Furthermore, we can observe that there is few instances of `mapping` tags in commits from CK group, nearly 12%. In contrast, commits modifying CK from the remaining groups ([AM, CK], [CK, FM], [AM, CK, and FM]) have at least 50% of changes related to `mapping`. Moreover, looking towards FM space, Table 7 presents tags in commits that change Kconfig file. According our results shown in Table 7, we can observe that 50 commits modifying all SPL spaces (AM CK FM group) present `feature` tag, representing 64%. Among these 50 commits, 45 add a feature declaration in Kconfig. On the other hand, between commits which modify only FM, few present feature declaration, around 8%. Furthermore, `depends` tags have the most occurrences in the follow groups: [FM], [AM and FM], and [CK and FM], except for the [AM, CK, and FM] group.

4.1.2.1 Existing Safe Templates in Remaining Commits

Through our manual analysis, we identify evolution scenarios in Remaining Commits which could be expressed in some already existing templates available on catalogue online (GROUP, 2018 (accessed november, 2018)). These templates and its SPL spaces modified are exhibited in Table 8.

In the group which only modifies CK, we found one commit renaming the feature expression (*mappingR*), and hence, could be categorized as REPLACE FEATURE EX-

	CK		
	ifdef	mapping	build
GROUPS			
AM	—	—	—
CK	6 (4.9%)	15 (12.3%)	**80 (65.6%)**
FM	—	—	—
AM CK	23 (16.8%)	70 (51.1%)	53 (38.7%)
AM FM	—	—	—
CK FM	9 (34.6%)	16 (61.5%)	5 (19.2%)
AM CK FM	20 (25.7%)	62 (79.5%)	**9 (11.5%)**

Table 6 – Summary of tags from remaining commits which modify CK space

	FM				
	select	feature	menu	depends	default
GROUPS					
AM	—	—	—	—	—
CK	—	—	—	—	—
FM	7 (14.3%)	4 (8.2%)	2 (4.1%)	31 (63.3%)	7 (14.3%)
AM CK	—	—	—	—	—
AM FM	2 (13.4%)	4 (26.7%)	0 (0%)	6 (40%)	0 (0%)
CK FM	2 (7.7%)	14 (54%)	0 (0%)	13 (50%)	0 (0%)
AM CK FM	11 (14%)	50 (64%)	7 (9%)	15 (19.2%)	3 (3.8%)

Table 7 – Summary of tags from remaining commits which modify FM space

Templates	FM	CK	AM
Merge Assets		x	x
Split Asset		x	x
Feature Renaming	x		
Asset Name Renaming			x
Replace Feature Expression		x	
Simplify Feature Expression using the FM		x	

Table 8 – Existing Safe Templates identified in Remaining Commits and its respective modified spaces

PRESSION template (hash *aeff3ff468*).[2] Further, in groups which modifies CK and AM, we identify 3 instances of SPLIT ASSET template, supporting scenarios which split some asset and create a new mapping in CK (hashes: *627a606550*, *8f007e0b74*, *627a606550*).[3] Additionally, we also found 1 evolution scenario which match with MERGE ASSETS template (hash: *16a4ed44fd*), which is like a inverse way of SPLIT ASSET template.[4] Furthermore, we categorize some commits with ASSET NAME RENAMING template (hashes: *1e6ad8b904*, *8c73c7bb12*, *ca032af983*, *8240e4ca5d*).[5]

In fact, it is not easy to assure that queries capture more than one template for each commit correctly. However, our manual analysis identified some evolution scenarios with templates combination. In `AM, CK, and FM changes` group, we observe one evolution scenario with a sequence of SPLIT ASSET, and ADD NEW FEATURE FROM EXISTING ARTIFACTS templates (hash *5d828031aa*) (BENBASSAT; BORBA; TEIXEIRA, 2016). This commit presents added assets derived from code extracted from other source files, further, a new feature is added with a mapping relating this feature with the extracted assets. There are commits (hash: *aca3b23d5c*, *c1d5b0f4a9*, *b29fad7082*) which should be categorized as three templates: FEATURE RENAMING, SIMPLIFY FEATURE EXPRESSION USING THE FM, and REFINE ASSET. These commits present scenarios with renamed config names (FEATURE RENAMING template), update of the feature expression in CK mapping (SIMPLIFY FEATURE EXPRESSION USING THE FM template), and then, the assets are refined due to its modification being related with the renaming feature expression in code.

Table 9 summarizes the results of the Soletta commits classification in templates. In automatic analysis phase, we classified commits through the queries in dataset. As results, from 2300 commits analyzed, 1800 was automatically classified as templates. On the other hand, manual analysis was performed in the 500 remaining commits, resulting in 13 commits classified as templates.

[2] https://github.com/solettaproject/soletta/commit/aeff3ff468
[3] https://github.com/solettaproject/soletta/commit/627a606550
[4] https://github.com/solettaproject/soletta/commit/16a4ed44fd
[5] https://github.com/solettaproject/soletta/commit/1e6ad8b904

Phase	**Evolution**	**Template**	**Commits classified as templates**
automatic analysis	partially	Remove assets	0 (0%)
		Add assets	0 (0%)
		Change asset	1,662 (72.26%)
		Change CK lines	35 (1.52%)
		Remove feature	2 (0.09%)
	safe	Add new optional feature	56 (2.44%)
		Add feature (no CK and AM)	5 (0.21%)
		Remove unused assets	6 (0.26%)
		Add unused assets	34 (1.48%)
manual analysis	safe	Replace feature expression	1 (0.04%)
		Split asset	3 (0.13%)
		Merge asset	1 (0.04%)
		Asset name Renaming	4 (0.17%)
		Split Asset and Add New Feature from Existing Artifacts	1 (0.04%)
		Feature Renaming, Simplify Feature Expression using the FM, and Refine Asset	3 (0.13%)

Table 9 – Final results of commits classified as templates in Soletta history

4.2 LINUX ANALYSIS

In the same way as Soletta, we also use FEVER to make automatic analysis over 13,288 commits of Linux Kernel, between versions 3.12 and 3.13, ranging from February/2013 to December/2013. Table 10 shows the summary of results of the Linux dataset extracted from Neo4j queries. As result, from all commits, 11,377 (85.62%) were captured by queries. On the following, we show the amount of commits captured for each query based on safe and partially safe templates:

Likewise Soletta, in our sample of Linux there is no instance of the Remove Assets template. In the same way, in Linux, Change Asset template also has the most

Linux v3.12-3.13	Commits	%
Total	13,288	100%
Templates	11,377	85.62%
Not captured as Template	1,911	14.38%
Excluded	15	0.11%
Remaining Commits	1,896	14.27%

Table 10 – Automatic Results in Linux v3.12-v3.13

Evolution	Template	Commits
partially	Remove assets	0 (0%)
	Add assets	4 (0.03%)
	Change asset	11,211 (84.36%)
	Change CK lines	17 (0.13%)
	Remove feature	12 (0.09%)
safe	Add new optional feature	78 (0.59%)
	Add feature without change CK and AM	9 (0.07%)
	Remove unused assets	8 (0.06%)
	Add unused assets	38 (0.29%)

Table 11 – Commits returned by queries derived from templates in Linux v3.12-v3.13

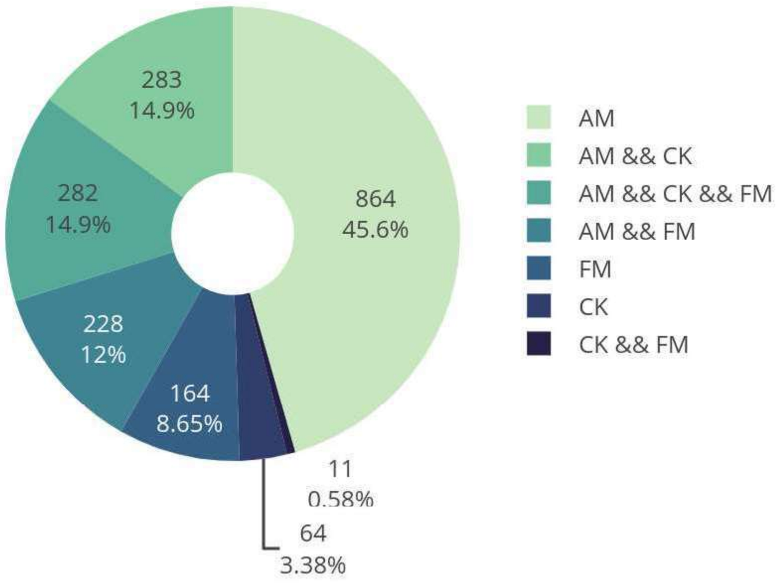


Figure 24 – 1,896 Remaining Commits of Linux v3.12-3.13

frequency between commits, around 84%.

From the 1,911 remaining commits, we excluded 15 instances not correctly mined by Repodriller. Between the excluded commits, 13 present several changes, such as the *5d43889c07* evolution scenario.[6] We guess that Repodriller does not mine correctly commits with more than 5k modified files. In other hand, the two remaining commits present no files modified (hashes: *9fbeace73c*, *1c5054d9e3*). Thus, we analyzed 1,896 remaining commits and extracted information about the modified spaces, which we plot in Figure 24. Differently from Soletta, Linux presents almost half of the remaining commits with evolution scenarios which only modify AM. The fewest occurrences are related to evolution scenarios which modify both CK and FM. Similar to Soletta results, in Linux also there are many changes which modify CK and AM at the same commit. Finally, we also plot a

[6] https://github.com/torvalds/linux/commit/5d43889c07

timeline of Linux templates, without Change Asset, exhibited in Figure 25. According to Figure 25, Add new Optional Feature template instances occurs more in the latest months of the release, different from Soletta. Moreover, Remove unused Assets template also presents more instances in the end of the release. Furthermore, such as in Soletta, the messages of these commits express that deleted files were useless.

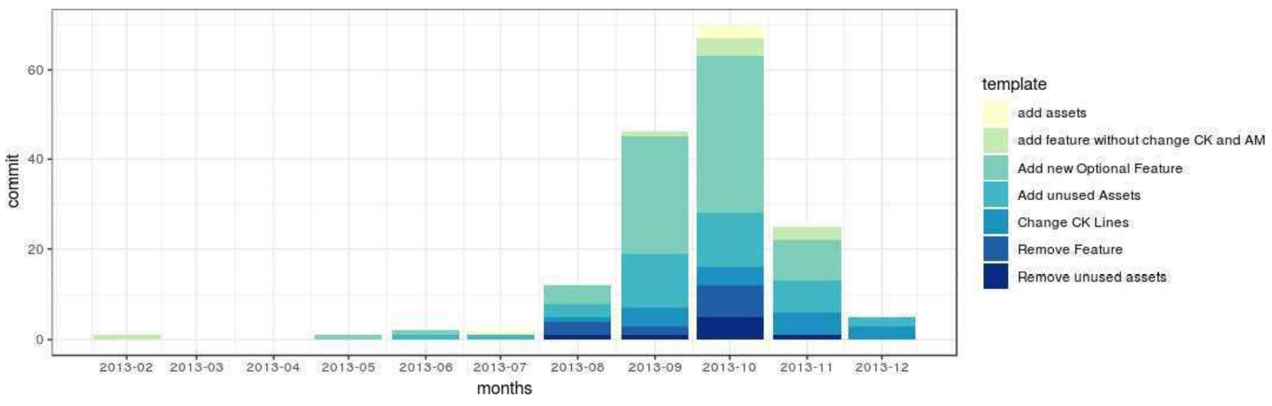


Figure 25 – Timeline of Linux Templates without Change Asset

4.3 DISCUSSION

After classifying the entire sample of the Soletta commits, we also establish the evolution type for each one of them. For the 1,800 commits classified as templates, the evolution type only depends on the template specification. For instance, commits classified as Add new Optional Feature are *safe*, while commits classified as Remove Feature are *partially safe*. For the remaining ones we classified according to our manual analysis. Finally, we report our results to our research questions:

4.3.1 RQ1: How changes are distributed in terms of safe and partially safe during the history of a software product line?

Figure 26 illustrates a *timeline* from our sample of 2,300 commits according to the evolution type, safe or partially safe. As a result, *partially safe* changes are more expressive during the entire Soletta history which we evaluated, representing 91.3% of the commits, while *safe* changes account for 8.7% only. These partially safe scenarios are mostly occurrences of the Change Asset template. Since we consider all instances of this template as *partially safe*, there might be instances where assets are changed in a *safe* way, which would be consistent with the Refine Asset safe evolution template. Thus, it could be the case that there are more safe evolution scenarios in such commits. Nonetheless, we do not believe that this would drastically change the numbers. Some scenarios are trivially safe, even through manual analysis, such as *rename* cases, for example. However, most of the changes are not easy to classify, so in such cases, we classified the change as partially

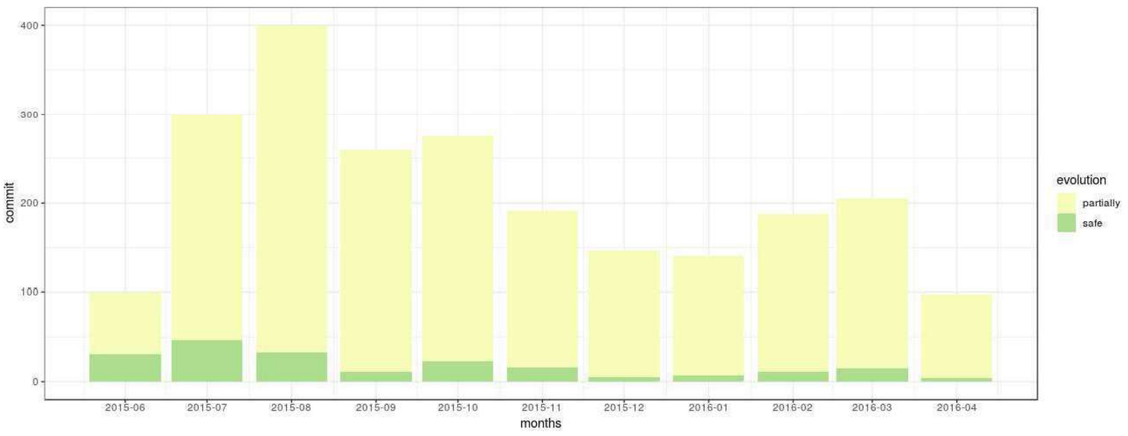


Figure 26 – Timeline Safe vs Partially Safe Evolution from 2,300 Soletta commits

safe evolution. Regarding the Linux analysis, from 11,377 commits captured by queries, around 85% represent partially safe evolution scenarios. Furthermore, most of those partially safe evolution scenarios were classified as Change asset template. In contrast, only 0.59% of commits present Add new optional feature. We believe that this low rate is due to the maturity of Linux Kernel. Changes tend to improve the code rather than add new functionality.

Can we observe patterns on how evolution happen?

In terms of evolution patterns for the remaining commits from Soletta, as aforementioned, commits with both AM and CK changes present the highest occurrence rate. After classifying the remaining commits in terms of *change type* and *tags*, we observe some patterns according to the modified spaces. For instance, in evolution scenarios which only modify the CK, the most common change is related to `build`. In contrast, evolution scenarios which modify both CK and AM present more changes related to `mapping` than `build` changes. This indicates the need for deriving templates to support these kinds of changes. With respect to the Linux Kernel analysis, we can observe that at least 30% of the remaining commits modify the CK. Likewise Soletta, there are several commits modifying both AM and CK among remaining commits.

4.3.2 RQ2: How often templates cover these real scenarios?

As we mentioned before, existing templates and their respective queries cover 78% of commits from Soletta sample. In fact, we did not use all of the available templates to perform queries. Through our manual analysis, we identify that some of the remaining commits are also classified as an already existing template. For instance, there were some evolution scenarios in the *AM and CK changes* group which we classified using existing templates, which were not feasible to express in queries, such as Split Asset and Merge Assets. These templates include asset refinement as one of the preconditions, which is an information that cannot be queried against the database. On Linux, 85.62% of commits were covered by existing templates, and the remaining one are not covered by any template.

Do we need to derive new templates?

Despite most evolution scenarios being classified as templates, a reasonable number were still unclassified. In fact, most of the existing templates in the catalog (GROUP, 2018 (accessed november, 2018)) focus on changes solely to the FM, or co-evolution of FM and other spaces. However, when evaluating the remaining commits from Soletta and Linux, we observe that some recurrent scenarios can be deeper analyzed to further derive new templates. Soletta results show that there is a lack of templates considering changes to FM and CK, such as adding feature dependencies in Kconfig, and simultaneously, changing the mapping in the Makefile. Also, there is a reasonable number of evolution scenarios that change the mapping in the Makefile and add some assets (AM).

4.4 THREATS TO VALIDITY

As any case study, our exploratory work also presents threats to validity. This section discusses some of those threats in what follows, according to guidelines from Runeson et al. (RUNESON et al., 2012).

4.4.1 Construct Validity

Evolution scenarios consists in changes made by developers. In our study, we consider each commit as a evolution scenario, due this type of contribution commonly represents an instance of change in which the developer goals perform. This type of concept of evolution could result bias in our analysis, so we define that as a construct threat. In other hand, our results shows that our metric, which each commit is one evolution scenario, works well in the SPL analyzed, because our results shows that most of those commits are covered by templates. However, we dispose our methodology, aiming to make the analysis with SPL with other metric of contributions, such as a sequence of commits, for example.

The knowledge about safe templates also can be a construct threat. Thus, if some developer were aware about the use of the evolution patterns, this could bias the results about the safe templates. We intend to check this issues with the Soletta developers in future works.

4.4.2 Internal Validity

We define as a first internal threat the tools used in our study. The FEVER tool was developed based on the Linux structure, and Soletta is a project that follows a similar structure, which makes the tool to work as expected. To analyze that the queries that we created based on this structure are not biased, we manually checked each one of the commits yielded by the queries. Nevertheless, a single-person analysis could also introduce bias in the results. This way, we consider the manual analysis as a second *internal threat.*

Aiming to mitigate this threat, we had another author reviewing the results to increase the confidence of our analysis. Thus, all results were checked and analyzed in pair. To solve doubts in the consensus phase, for some of the analyzed commits, we discussed with a third researcher that supervised the consensus phase. There was no strong disagreement. We also make available in our online appendix (GOMES, 2019 (accessed february, 2019)) the study package, including the commits covered by templates and also the dataset with the remaining commits tagged by keywords and change types.

4.4.3 External Validity

Our study analyzed only one project, and we consider this as an *external threat.* However, the Soletta structure is similar to other SPL projects which use Kconfig to manage variability and the C language for implementation. We have conducted a preliminary study with commits from the Linux kernel and the automated classification reveals similar numbers. We also make available our methodology, allowing future analysis in other projects to confirm our results.

4.4.4 Reliability

Runeson et al. (RUNESON et al., 2012) presents the Reliability concept that concerns to what extent the dependency of the work results by the research authors. To mitigate the bias, we make available our methodology to yield further analysis from other researchers' perspectives. Furthermore, the study materials are available in the online appendix, aiming to further reproducibility and replicability of our work.

5 CONCLUSIONS

In Software Engineering, Software Product Line is a technique to provide reuse and customization in a systematic way. SPL is often represented as set of features which can be reused, and according to their constraints, forming distinct valid product. This approach brings many advantages, such as reduced time-to-market, reduced costs, and quality improvement. An SPL is usually structured in three spaces: Feature Model, Configuration Knowledge and Asset Mapping.

Due to the reusable structure of SPL, one single feature can be spread across a range of valid configurations. So, performing changes in SPL might be error-prone, because a simple modification can impact several products. Hence, these questions induce previous works to investigate SPL evolution, introducing theories that allow reasoning about safe and partially safe evolution, formalized through the refinement theory (). Later, researchers suggested templates based on these theories to support developers during evolution scenarios. However, there is no evidence about the interplay of both safe and partially safe evolution templates in the context of real world SPLs. Therefore, these issues motivated us perform an empirical study about safe and partially safe evolution in SPL. For that, we use a real-world project: Soletta, a framework to support writing devices for Internet of Things. We analyze 2,300 commits, in a range of 1 year of project (jun/2015 to apr/2016).

First, we use the FEVER tool to extract information for each commit (or evolution scenario) from Soletta repository. The FEVER output is stored in a Neo4j graph dataset. After mining the repository we choose some safe and partially safe templates to express in queries to capture the commits which matches with these patterns. However, neither all templates could be precisely captured by queries, so we choose nine templates (four safe, and five partially safe). We obtain as results that 78.3% (1,800 occurrences) of commits are covered by templates, and the remaining 21.7% (500 occurrences) were not automatically classified. From 1,800 commits classified as templates, we verify that most of the occurrences is referring to Change Asset template, as expected, representing around 72% of evolution scenarios. Surprisingly, there are only two instances of Remove Feature template. Moreover, there is no instances of both Add Assets and Remove Assets partially safe templates.

Later, we analyzed the 500 commits not covered by templates. Initially we group these remaining commits according the changed SPL spaces in seven distinct groups: [AM], [CK], [FM], [FM, CK], [FM, AM], [CK, AM], and [FM, CK, AM]. Between these groups, commits which modifies both CK and AM occurs more frequently, around 27.4%. As a unexpected result, commits which change only the CK are the second most common occurrence, representing 24.4% from all remaining commits. In contrast, commits which present changes in both FM and AM at same time have the fewest instances, around 3%.

We define some tags based on characteristics of each space (FM: Kconfig; CK: Makefile; AM: assets) aiming to categorize the changes presented in commits. Furthermore, we also categorize the type of change which varies between *modify, add, remove, move, rename*, to relate them with the tags. As result, commits which only modify the FM present changes related to `depends on` clause. In other hand, in CK group commits, most of evolution scenarios referring to build changes. However, several scenarios which change CK with AM or FM spaces at same time, present modification in `mapping`. Through our manual analysis in remaining commits, we also identify some existing templates different from those we used to capture by queries: SPLIT ASSET, MERGE ASSETS, ASSET NAME RENAMING, REFINE ASSET, FEATURE RENAMING, and SIMPLIFY FEATURE EXPRESSION USING THE FM.

Finally, we classify the commits analyzed from Soletta as safe or partially safe. Commits captured by queries were classified according the specification of each template, for instance, Change Assets instances were classified as partially safe. We classify manually remaining commits in conservative approach, so, simple scenarios such as rename cases, we classify as safe evolution, and remaining ones as partially safe. As result, most of the evolution scenarios, around 91% of commits, do not preserve the behavior of previous SPL (partially safe evolution scenarios), and the remaining (around 9%) are classified as safe evolution scenarios.

As a preliminary study, we also make automatic analysis in 13,288 commits from Linux Kernel, between versions 3.12 and 3.13. Such as Soletta, most of the commits were captured by queries, and therefore referred to some existing template (11,377 instances, around 85% from all commits). In the 1,911 Remaining Commits, most of changes only modify the AM, around 45% (864 commits). In contrast, there are few instances which modify both CK and FM, 0.58% (11 commits). Moreover, evolution scenarios with changes to all SPL spaces represent 14.9% of remaining commits (282 commits).

5.1 CONTRIBUTIONS

This section presents the contributions from our empirical study, and we list them as follows:

- Improve understanding about safe evolution and partially safe evolution during the SPL life-cycle;
- Derive new queries representing safe evolution templates which capture real evolution scenarios during the SPL history;
- A methodology for manually analyzing changes that might reveal novel templates for expressing evolution scenarios;

- Availability of our data, tools and scripts used in our study for further replication and deeper analysis;
- Insights about evolution in SPL yielding future development of tools which could supports SPL developers to perform their changes;

5.2 RELATED WORK

The work reported here is based on existing studies over SPL refactoring and evolution (ALVES et al., 2006; BORBA, 2011; BORBA; TEIXEIRA; GHEYI, 2012; NEVES et al., 2015b; BENBASSAT; BORBA; TEIXEIRA, 2016; SAMPAIO; BORBA; TEIXEIRA, 2016). Alves et al. (ALVES et al., 2006) extend refactoring concepts to the SPL context, proposing a catalogue of FM refactorings.

The notion of *Safe Evolution* discussed here first appeared with a refactoring focus (BORBA, 2011), illustrating different kinds of refactoring transformation templates that can be useful for deriving and evolving product lines. Borba et al. (BORBA; TEIXEIRA; GHEYI, 2012) mechanized and generalized the initial proposal into a refinement theory, introducing and proving soundness for a number of SPL transformation templates. Based on this theory, with the goal of guiding developers in possible refinement scenarios, Neves et al. (NEVES et al., 2015b) and Benbassat et al. (BENBASSAT; BORBA; TEIXEIRA, 2016) propose template catalogues to abstract *Safe Evolution* scenarios.

Sampaio et al. (SAMPAIO; BORBA; TEIXEIRA, 2016) extends the refinement theory with the concept of *partial refinement*, establishing the concept of Partially Safe Evolution. This concept, as discussed, allows supporting changes that only preserve the behavior of a subset of the existing products. Our study differs from these previous studies by focusing on both safe and partially safe evolution templates, to understand the distribution of such changes throughout the evolution history of an SPL. Sampaio also goes beyond our analysis of change assets, by considering the commit messages to further classify such scenarios as refactorings or not. We plan to follow a similar strategy in future works.

Montalvillo et al. perform a mapping study (MONTALVILLO; DíAZ, 2016) which classified studies related to evolution in SPLs, and their results shows that few studies focus on identifying changes in SPLs. Our work categorizes changes performed during the SPL evolution history life-cycle, measuring how often templates cover changes in real projects, and also characterizing changes not mapped to existing templates using change types and tags.

Dintzner et al. (DINTZNER; DEURSEN; PINZGER, 2013) present a tool named *FMDiff* to automatically analyze differences in Linux Kconfig models. The change categories are specific to structures found in Kconfig specifications, such as feature dependency changes. This tool could be used to cross-check our manual tag analysis of changes to the FM. Dintzner et al. (DINTZNER; DEURSEN; PINZGER, 2018) also developed the FEVER tool, which we use in our evaluation, that enables the commit analysis of Kconfig-based systems, extracting feature-oriented changes from the commits. In our work, we go beyond what FEVER is able to extract, since we also want to classify commits as safe and partially safe.

Passos et al. (PASSOS et al., 2016) perform a study analyzing how evolution occurs on the Linux kernel. Their study is focused on changes that involve feature addition or removal. As a result, they also provide a pattern catalogue, similar to the templates we discuss here. However, their focus is not on categorizing such patterns as safe or partially safe. In contrast, our study analyzed all of the commits, regardless of specific changes to a particular SPL space such as the FM.

Bürdek et al. (BÜRDEK et al., 2016) propose an approach to document and classify changes in feature diagrams using a logic-based formal framework. They provide a catalogue describing structural changes in feature models. Different from this study, our work analyze evolution scenarios and how those changes affect the three SPL spaces, not only focusing in the FM. Moreover, our intention is also on classifying scenarios into safe or partially safe, according to the kind of change.

Kröher et al. (KRöHER; GERLING; SCHMID, 2018) perform a study to understand the intensity of variability-related changes to the Linux kernel. They measure how often changes occur in FM, AM, and CK, and how often do those changes are related to variability information inside these artifacts. Our work also investigate the intensity of changes for the remaining commits, going into detail of what has been changed in the tag classification. However, our focus is on classifying evolution scenarios into safe or partially safe and to use the tags as a way to derive new templates in the future. Nonetheless, their tool could be a complementary tool to our analysis, and we could cross-check our results to see if the same patterns that occur in Linux also hold for Soletta.

5.3 FUTURE WORK

Our work consists in an empirical study to characterize evolution of Software Product Lines, in terms of safe and partially safe. Through our study, we identify if templates are being used in practical scenarios. As expected, not all commits is expressed as template. So, we also categorize the type of change of the evolution scenarios which are not correspondent with templates. Furthermore, we make available our data, scripts, and methodology used to perform this work and classify the evolution scenarios. Although we make an classification of a SPL evolution as a whole and its characteristics, we identify some possibilities to improve our study which are described below:

- From all 2300 commits analyzed, 1662 instances only modify assets, in other words, there are 1662 commits classified as *change asset* template. So, this motivates us to perform a deeper analysis about the changes occurred in assets and its impacts;

- Moreover, we intend to explore our classification of `change type` and `tags` to derive new templates;
- As expected, most of commits are classified as partially safe evolution. Therefore, we intend to investigate thoroughly such evolution scenarios, aiming to define the subset of products affected by the changes;
- We also intend to analyze whether existing templates for individually changing FM and CK can be leveraged to improve the classification of remaining commits;
- Finally, from the previous issues and improvement of our work specified above, we aim to provide some tool to support developers to execute their changes in the SPL.

REFERENCES

ALVES, V.; GHEYI, R.; MASSONI, T.; KULESZA, U.; BORBA, P.; LUCENA, C. Refactoring product lines. In: *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*. New York, NY, USA: ACM, 2006. (GPCE '06), p. 201–210. ISBN 1-59593-237-2. Disponível em: <http://doi.acm.org/10.1145/1173706.1173737>.

APEL, S.; BATORY, D.; KSTNER, C.; SAAKE, G. *Feature-Oriented Software Product Lines: Concepts and Implementation*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2016. ISBN 3662513005, 9783662513002.

BASILI, V. R. *Software Modeling and Measurement: The Goal/Question/Metric Paradigm*. College Park, MD, USA, 1992.

BENBASSAT, F.; BORBA, P.; TEIXEIRA, L. Safe evolution of software product lines: Feature extraction scenarios. In: *2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*. [S.l.: s.n.], 2016. p. 11–20.

BORBA, P. An introduction to software product line refactoring. In: *Proceedings of the 3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering III*. Berlin, Heidelberg: Springer-Verlag, 2011. (GTTSE'09), p. 1–26. ISBN 3-642-18022-1, 978-3-642-18022-4. Disponível em: <http://dl.acm.org/citation.cfm?id=1949925.1949927>.

BORBA, P.; TEIXEIRA, L.; GHEYI, R. A theory of software product line refinement. *Theoretical Computer Science*, v. 455, p. 2 – 30, 2012. ISSN 0304-3975. International Colloquium on Theoretical Aspects of Computing 2010. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0304397512000679>.

BÜRDEK, J.; KEHRER, T.; LOCHAU, M.; REULING, D.; KELTER, U.; SCHÜRR, A. Reasoning about product-line evolution using complex feature model differences. *Automated Software Engineering*, v. 23, n. 4, p. 687–733, Dec 2016. ISSN 1573-7535. Disponível em: <https://doi.org/10.1007/s10515-015-0185-3>.

CLEMENTS, P.; NORTHROP, L. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002. (SEI series in software engineering). ISBN 9780201703320. Disponível em: <https://books.google.com.br/books?id=tHGFQgAACAAJ>.

DINTZNER, N.; DEURSEN, A. V.; PINZGER, M. Extracting feature model changes from the linux kernel using fmdiff. In: *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*. New York, NY, USA: ACM, 2013. (VaMoS '14), p. 22:1–22:8. ISBN 978-1-4503-2556-1. Disponível em: <http://doi.acm.org/10.1145/2556624.2556631>.

DINTZNER, N.; DEURSEN, A. van; PINZGER, M. Fever: Extracting feature-oriented changes from commits. In: *Proceedings of the 13th International Conference on Mining Software Repositories*. New York, NY, USA: ACM, 2016. (MSR '16), p. 85–96. ISBN 978-1-4503-4186-8. Disponível em: <http://doi.acm.org/10.1145/2901739.2901755>.

DINTZNER, N.; DEURSEN, A. van; PINZGER, M. Fever: An approach to analyze feature-oriented changes and artefact co-evolution in highly configurable systems. *Empirical Software Engineering*, v. 23, n. 2, p. 905–952, Apr 2018. ISSN 1573-7616. Disponível em: <https://doi.org/10.1007/s10664-017-9557-6>.

GOMES, K. *Vamos 2019 - Characterizing safe and partially safe evolution scenarios in product lines: An Empirical Study.* [S.l.], 2019 (accessed february, 2019). <http://www.cin.ufpe.br/~kgmg/msc>.

GROUP, S. *Templates for Software Product Line Evolution.* [S.l.], 2018 (accessed november, 2018). <https://github.com/spgroup/pl-refinement-templates-catalog/blob/master/templatescatalog.pdf>.

HEIDER, W.; VIERHAUSER, M.; LETTNER, D.; GRüNBACHER, P. A case study on the evolution of a component-based product line. In: *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture.* [S.l.: s.n.], 2012. p. 1–10.

KANG, K. C.; COHEN, S. G.; HESS, J. A.; NOVAK, W. E.; PETERSON, A. S. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* [S.l.], 1990.

KERNEL. *Kconfig Language.* [S.l.], 2018 (accessed August, 2018). Disponível em: <https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt>.

KRöHER, C.; GERLING, L.; SCHMID, K. Identifying the intensity of variability changes in software product line evolution. In: *Proceedings of the 22Nd International Systems and Software Product Line Conference - Volume 1.* New York, NY, USA: ACM, 2018. (SPLC '18), p. 54–64. ISBN 978-1-4503-6464-5. Disponível em: <http://doi.acm.org/10.1145/3233027.3233032>.

KRÜGER, J.; GU, W.; SHEN, H.; MUKELABAI, M.; HEBIG, R.; BERGER, T. Towards a better understanding of software features and their characteristics: A case study of marlin. In: *VaMoS.* [S.l.: s.n.], 2018.

LOTUFO, R.; SHE, S.; BERGER, T.; CZARNECKI, K.; WaSOWSKI, A. Evolution of the linux kernel variability model. In: *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond.* Berlin, Heidelberg: Springer-Verlag, 2010. (SPLC'10), p. 136–150. ISBN 3-642-15578-2, 978-3-642-15578-9. Disponível em: <http://dl.acm.org/citation.cfm?id=1885639.1885653>.

MONTALVILLO, L.; DíAZ, O. Requirement-driven evolution in software product lines: A systematic mapping study. *Journal of Systems and Software*, v. 122, p. 110 − 143, 2016. ISSN 0164-1212. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0164121216301510>.

NEVES, L.; BORBA, P.; ALVES, V.; TURNES, L.; TEIXEIRA, L.; SENA, D.; KULESZA, U. Safe evolution templates for software product lines. *J. Syst. Softw.*, Elsevier Science Inc., New York, NY, USA, v. 106, n. C, p. 42–58, ago. 2015. ISSN 0164-1212. Disponível em: <http://dx.doi.org/10.1016/j.jss.2015.04.024>.

NEVES, L.; BORBA, P.; ALVES, V.; TURNES, L.; TEIXEIRA, L.; SENA, D.; KULESZA, U. Safe evolution templates for software product lines. *Journal of*

Systems and Software, v. 106, p. 42 – 58, 2015. ISSN 0164-1212. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0164121215000801>.

PASSOS, L.; CZARNECKI, K. A dataset of feature additions and feature removals from the linux kernel. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. New York, NY, USA: ACM, 2014. (MSR 2014), p. 376–379. ISBN 978-1-4503-2863-0. Disponível em: <http://doi.acm.org/10.1145/2597073.2597124>.

PASSOS, L.; CZARNECKI, K.; WASOWSKI, A. Towards a catalog of variability evolution patterns: The linux kernel case. In: *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*. New York, NY, USA: ACM, 2012. (FOSD '12), p. 62–69. ISBN 978-1-4503-1309-4. Disponível em: <http://doi.acm.org/10.1145/2377816.2377825>.

PASSOS, L.; TEIXEIRA, L.; DINTZNER, N.; APEL, S.; WASOWSKI, A.; CZARNECKI, K.; BORBA, P.; GUO, J. Coevolution of variability models and related software artifacts. *Empirical Software Engineering*, v. 21, n. 4, p. 1744–1793, Aug 2016. ISSN 1573-7616. Disponível em: <https://doi.org/10.1007/s10664-015-9364-x>.

POHL, K.; BöCKLE, G.; LINDEN, F. J. v. d. *Software Product Line Engineering: Foundations, Principles and Techniques*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005. ISBN 3540243720.

RUNESON, P.; HOST, M.; RAINER, A.; REGNELL, B. *Case Study Research in Software Engineering: Guidelines and Examples*. 1st. ed. [S.l.]: Wiley Publishing, 2012. ISBN 1118104358, 9781118104354.

SAMPAIO, G.; BORBA, P.; TEIXEIRA, L. Partially safe evolution of software product lines. In: *Proceedings of the 20th International Systems and Software Product Line Conference*. New York, NY, USA: ACM, 2016. (SPLC '16), p. 124–133. ISBN 978-1-4503-4050-2. Disponível em: <http://doi.acm.org/10.1145/2934466.2934482>.

SHE, S.; LOTUFO, R.; BERGER, T.; WASOWSKI, A.; CZARNECKI, K. The variability model of the linux kernel. In: . [S.l.: s.n.], 2010. Null ; Conference date: 27-01-2010 Through 29-01-2010.

TEIXEIRA, L.; ALVES, V.; BORBA, P.; GHEYI, R. A product line of theories for reasoning about safe evolution of product lines. In: *Proceedings of the 19th International Conference on Software Product Line*. New York, NY, USA: ACM, 2015. (SPLC '15), p. 161–170. ISBN 978-1-4503-3613-0. Disponível em: <http://doi.acm.org/10.1145/2791060.2791105>.

APPENDIX A – SAFE AND PARTIALLY SAFE TEMPLATES USED AUTOMATICALLY THROUGH QUERIES IN OUR EMPIRICAL STUDY

A.1 SAFE TEMPLATES

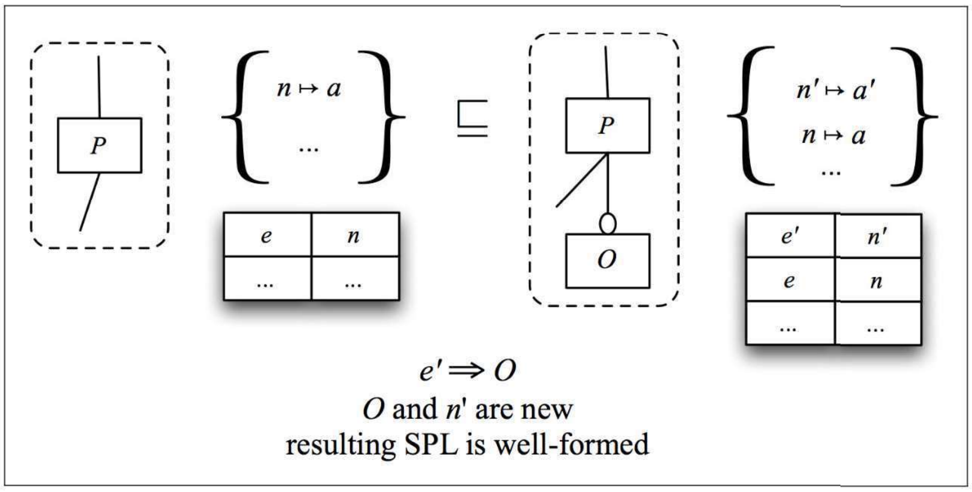


Figure 27 – Add new Optional Feature

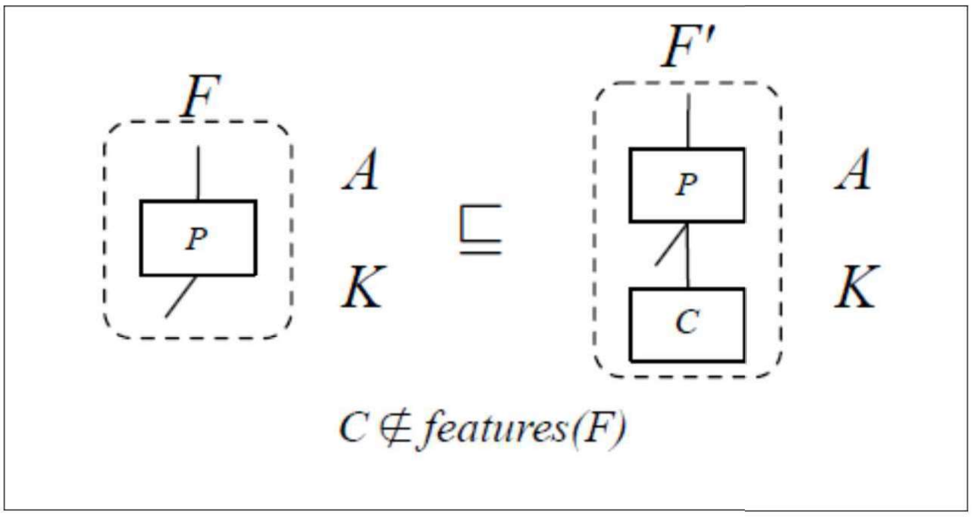


Figure 28 – Add any Feature without change CK and AM

$$\begin{array}{ccc} F & & F \\ A & \sqsubseteq & A \oplus m \\ K & & K \end{array}$$

$$\forall\, n \in dom(m) \bullet n \text{ does not appear in K}$$

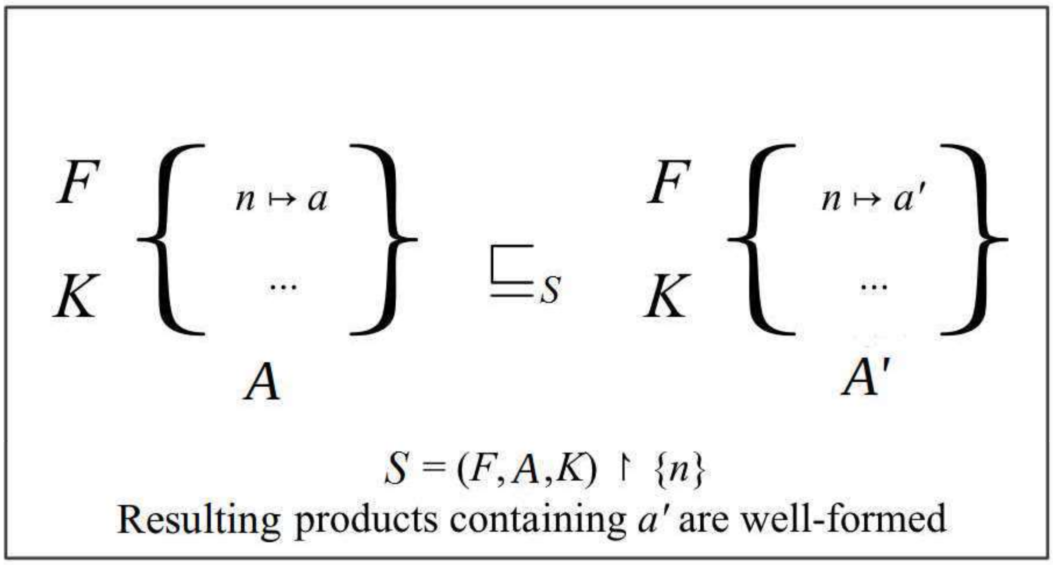
Figure 29 – Add unused Assets

$$\begin{array}{ccc} F & & F \\ A' \oplus m & \sqsubseteq & A' \\ K & & K \end{array}$$

$$\forall\, n \in dom(m) \bullet n \text{ does not appear in K}$$

Figure 30 – Remove unused Assets

A.2 PARTIALLY SAFE TEMPLATES

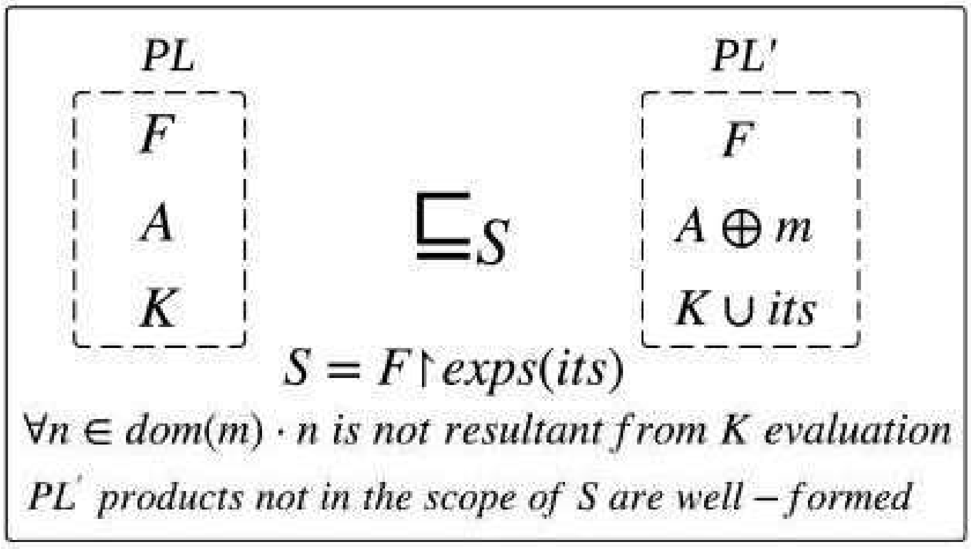


$F \left\{ \begin{array}{c} n \mapsto a \\ \ldots \end{array} \right\}_A K \sqsubseteq_S F \left\{ \begin{array}{c} n \mapsto a' \\ \ldots \end{array} \right\}_{A'} K$

$S = (F, A, K) \upharpoonright \{n\}$

Resulting products containing a' are well-formed

Figure 31 – Change Asset



PL

F

A

K

$\sqsubseteq_S$

PL'

F

$A \oplus m$

$K \cup its$

$S = F \upharpoonright exps(its)$

$\forall n \in dom(m) \cdot n$ is not resultant from K evaluation

PL' products not in the scope of S are well − formed

Figure 32 – Add Assets

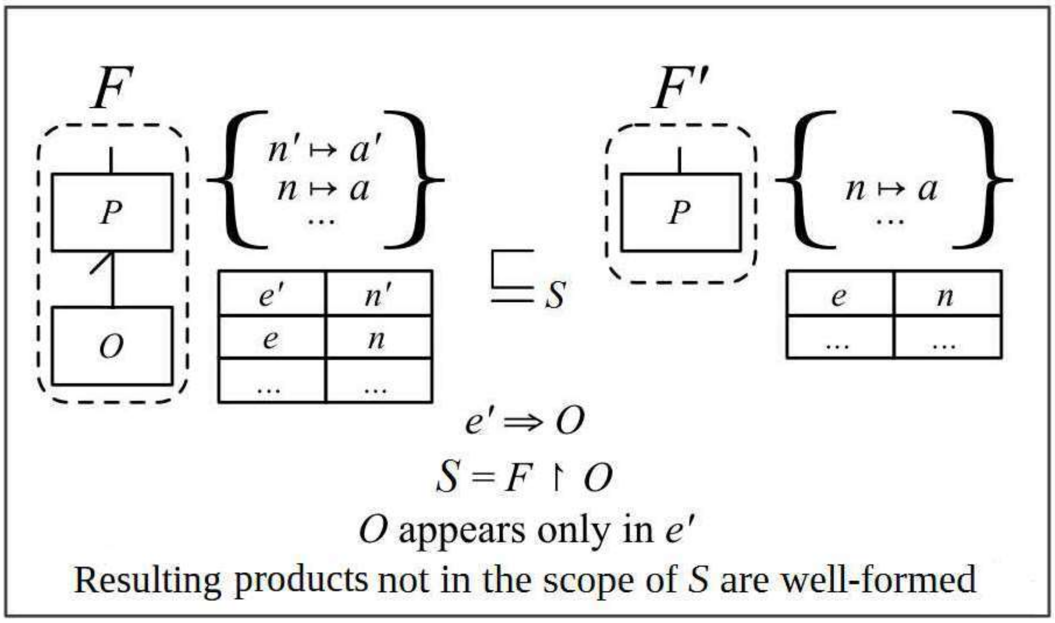

F
P
O
$\left\{ \begin{matrix} n' \mapsto a' \\ n \mapsto a \\ \ldots \end{matrix} \right\}$
e' | n'
e | n
... | ...
$\sqsubseteq_S$
F'
P
$\left\{ \begin{matrix} n \mapsto a \\ \ldots \end{matrix} \right\}$
e | n
... | ...
$e' \Rightarrow O$
$S = F \upharpoonright O$
O appears only in e'
Resulting products not in the scope of S are well-formed

Figure 33 – Remove Feature

APPENDIX B – SAFE TEMPLATES CAPTURED THROUGH MANUAL ANALYSIS IN OUR EMPIRICAL STUDY

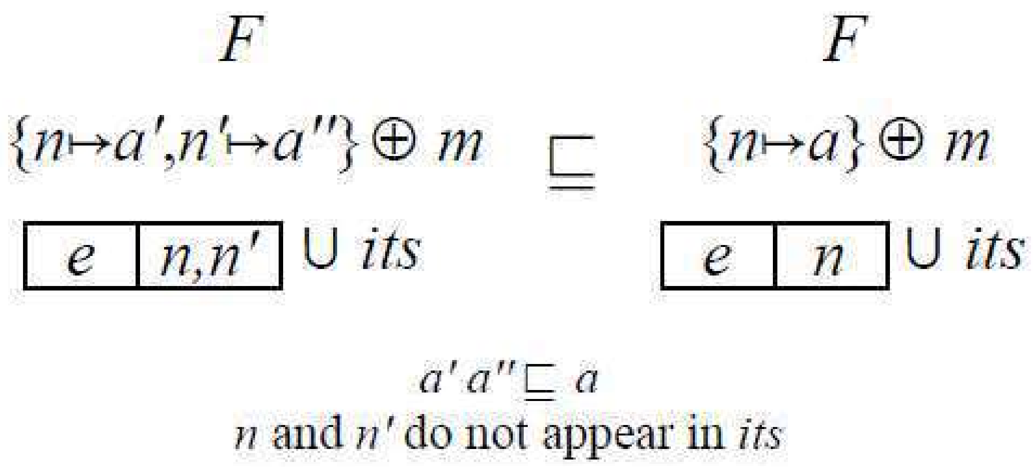

F
F
$\{n \mapsto a', n' \mapsto a''\} \oplus m$
$\sqsubseteq$
$\{n \mapsto a\} \oplus m$
e | n, n' $\cup$ its
e | n $\cup$ its
$a' a'' \sqsubseteq a$
n and n' do not appear in its

Figure 34 – Merge Assets

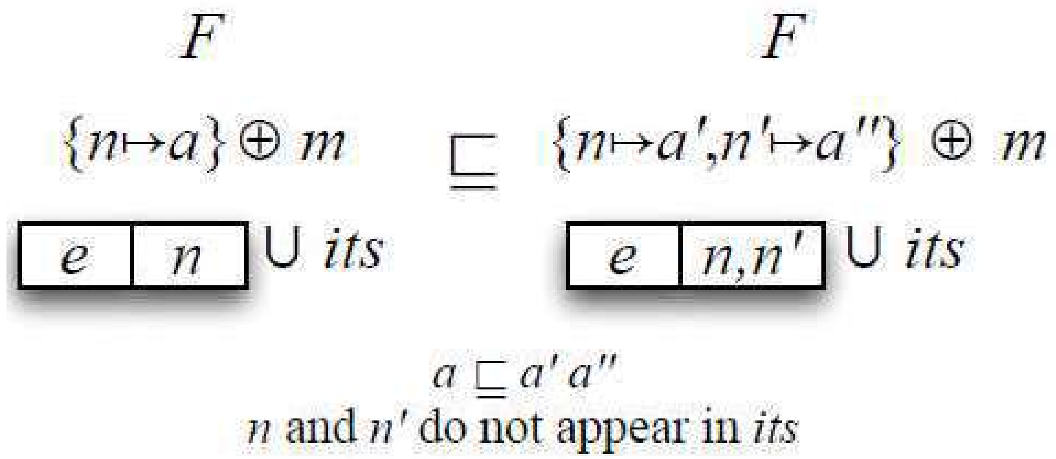

F
F
$\{n \mapsto a\} \oplus m$
$\sqsubseteq$
$\{n \mapsto a', n' \mapsto a''\} \oplus m$
e | n $\cup$ its
e | n, n' $\cup$ its
$a \sqsubseteq a' a''$
n and n' do not appear in its

Figure 35 – Split Assets

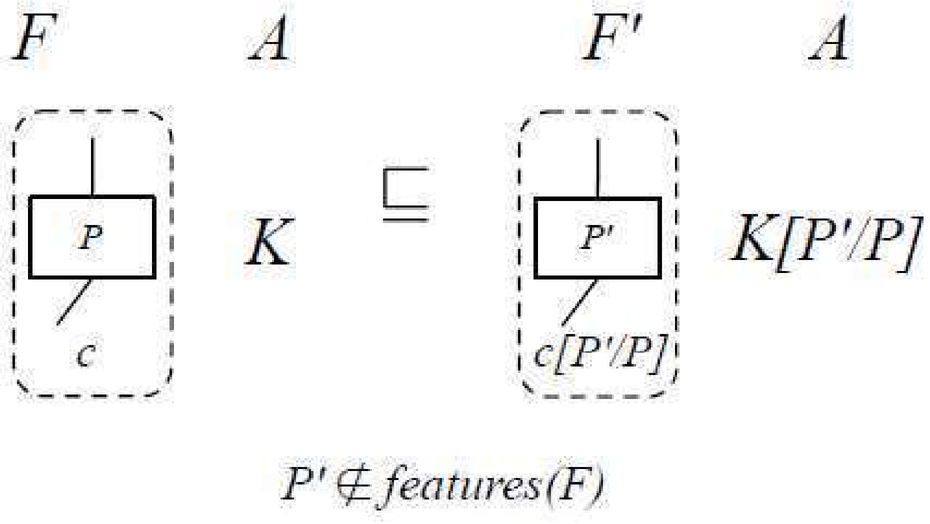


Figure 36 – Feature Renaming

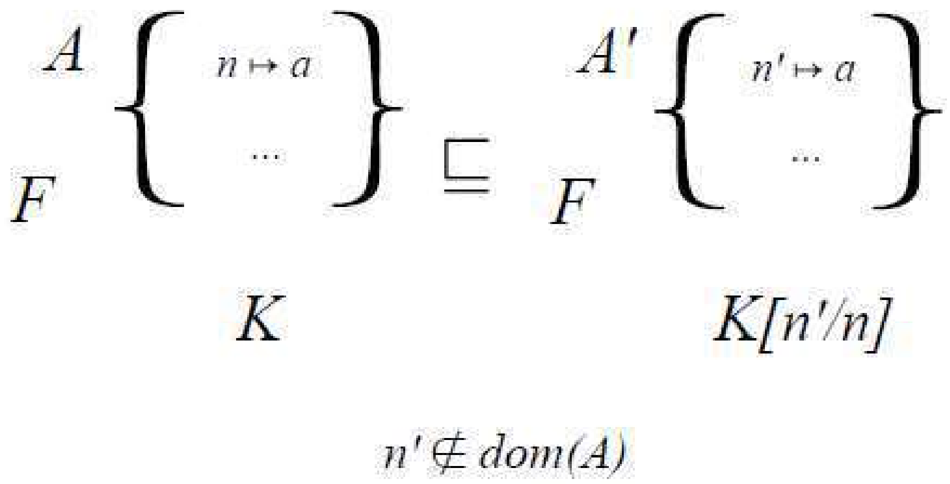


Figure 37 – Asset Name Renaming

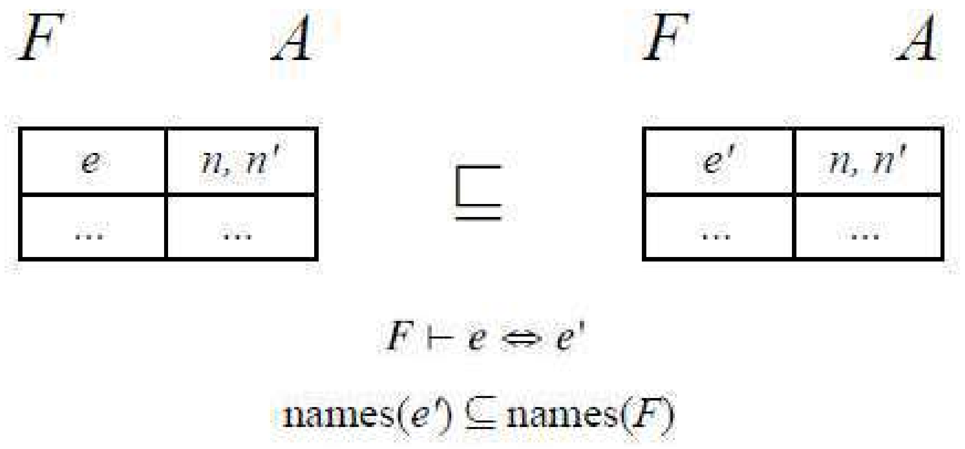


Figure 38 – Replace Feature Expression

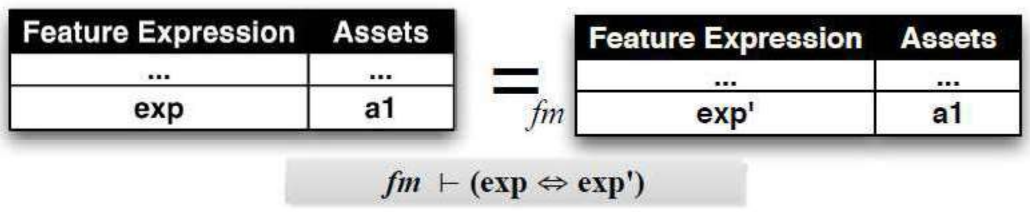


Figure 39 – Simplify Feature Expression using the FM

APPENDIX C – QUERIES DERIVED FROM TEMPLATES TO CAPTURE EVOLUTION SCENARIOS

Listing C.1 – (Partially safe) Change Asset Query

```
match (c: commit)
-->(a: ArtefactEdit {change: "MODIFIED"})
where
not (c)-->(:ArtefactEdit{change: "ADDED"}) and
not (c)-->(:ArtefactEdit{change: "REMOVED"}) and
not (c)-->(:ArtefactEdit{type: "vm"}) and
not (c)-->(:ArtefactEdit{type: "build"})
return distinct c.hash
```

Listing C.2 – (Partially safe) Remove Feature Query

```
match (file: ArtefactEdit)<-[:TOUCHES]-(c:commit)
-[:CHANGES_BUILD]->(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 =~
(".*" + substring (mapping.target, 0, length (mapping.target) -2) + ".*"))
return distinct c.hash
```

Listing C.3 – (Partially safe) Change CK Lines Query

```
match (c: commit) -[:CHANGES_BUILD ]->(:MappingEdit)
where
not (c) -->(:ArtefactEdit{type: "source"}) and
not (c) -->(:FeatureEdit )
return distinct c.hash
```

Listing C.4 – (Partially safe) Add Assets Query

```
match (file: ArtefactEdit )<--(c: commit )
-->(mapping: MappingEdit)
where
not (c)-->(:FeatureEdit) AND
not (c)-->(:MappingEdit {mapping_change: "MODIFIED"}) AND
not (c)-->(:MappingEdit {mapping_change: "REMOVED"}) AND
file.change = "ADDED" AND
mapping.target_change = "ADDED" AND
mapping.target_type = "COMPILATION_UNIT" AND
not (c)-->(:ArtefactEdit {type: "source", change: "MODIFIED"}) AND
not (c)-->(:ArtefactEdit {type: "source", change: "REMOVED"}) AND
file.name =~
(".*" + substring (mapping.target, 0, size(mapping.target) - 2) + ".*")
return distinct c.hash
```

Listing C.5 – (Partially safe) Remove Assets Query

```
match (file: ArtefactEdit )<-[:TOUCHES]-(c: commit)
-[:CHANGES_BUILD]->(mapping: MappingEdit)
where
not (c)-->(:FeatureEdit ) AND
not (c)-->(:MappingEdit {mapping_change: "MODIFIED"}) AND
not (c)-->(:MappingEdit {mapping_change: "ADDED"})
AND file.change =  "REMOVED" AND mapping.target_change = "REMOVED" AND
mapping.target_type = "COMPILATION_UNIT" AND
not (c)-->(:ArtefactEdit {type: "source", change: "MODIFIED"}) AND
not (c)-->(:ArtefactEdit {type: "source", change: "ADDED"}) AND
file.name =~
(".*"+ substring (mapping.target, 0, length (mapping.target) - 2) + ".*")
return distinct c.hash
```

Listing C.6 – (Safe) Add new Optional Feature Query

```
match (ae:ArtefactEdit {change: "ADDED"})<--(c:commit)
-[]->(f:FeatureEdit{change:"Add"})
-[]->(fd:FeatureDesc{optionality:"optional"})
where
(c)-->(:MappingEdit {feature: f.name}) and
(c)-->(:ArtefactEdit {type: "source", change: "ADDED"}) and not
(c)-->(:ArtefactEdit {type: "source", change: "MODIFIED"}) and not
(c)-->(:ArtefactEdit {type: "source", change: "REMOVED"})
or ae.name =~ ".*.jbp.*" or ae.name =~ ".*.json.*"
return distinct c.hash
```

Listing C.7 – (Safe) Add any Feature without change the CK and AM Query

```
match (c: commit) -->(f: FeatureEdit)
where
f.change =~ "Add" and
not (c)-->(:ArtefactEdit {type: "build"}) and
not (c)-->(:ArtefactEdit {type: "source"})
return distinct c.hash
```

Listing C.8 – (Safe) Remove unused Assets Query

```
match (c: commit )--> (a: ArtefactEdit {change: "REMOVED"})
where
not (c)-->(:ArtefactEdit {change: "MODIFIED"}) and
not (c)-->(:ArtefactEdit {change: "ADDED"}) and
not (:FeatureEdit) <--(c) and not (:MappingEdit) <--(c) and
not (c)-->(:ArtefactEdit {type: "build"}) AND not (c)-->(:ArtefactEdit {type: "vm"})
return distinct c.hash
```

Listing C.9 – (Safe) Add unused Assets Query

```
match (c: commit) --> (a: ArtefactEdit {change: "ADDED"})
where
not (c)-->(:ArtefactEdit {change: "MODIFIED"}) and
not (c)-->(:ArtefactEdit {change: "REMOVED"}) and
not (:FeatureEdit) <--(c) and not (:MappingEdit) <--(c) and
not (c)-->(:ArtefactEdit {type: "build"}) AND
not (c)-->(:ArtefactEdit {type: "vm"})
return distinct c.hash
```