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

Karine Gomes
Federal University of Pernambuco
Recife, Pernambuco
kgmg@cin.ufpe.br

Leopoldo Teixeira
Federal University of Pernambuco
Recife, Pernambuco
lmt@cin.ufpe.br

Thayonara Alves
Federal University of Pernambuco
Recife, Pernambuco
tpa@cin.ufpe.br

Márcio Ribeiro
Federal University of Alagoas
Maceió, Alagoas
marcio@ic.ufal.br

Rohit Gheyi
Federal University of Campina Grande
Campina Grande, Paraíba
rohit@dsc.ufcg.edu.br

ABSTRACT

Evolving software product lines is often error-prone. Previous works have proposed classifying product line evolution into safe or partially safe, depending on the number of products that have their behavior preserved after evolution. Based on these notions, it is possible to derive transformation templates that abstract common evolution scenarios, such as adding an optional feature. However, existing works are focused on evaluating either safe or partially safe templates. Hence, in this work we aim to characterize product line evolution as a whole, measuring to what extent the evolution history is safe compared to partially safe, to better understand how product lines evolve. We measure how often existing templates happen using 2,300 commits from an open-source product line. According to our study, 91.7% of the commits represent partially safe evolution scenarios. Our results also show that 1,800 of these commits can automatically be classified as instances of existing templates. Among these, commits that do not modify other variability-aware models, are the most frequent, accounting for 72.3% out of the total of commits. For the remaining 500 commits, we identify that 24.4% are related to changes in the configuration knowledge, that is, the file responsible for the mapping between features and code.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; **Software evolution**; • **General and reference →** *Empirical studies*;

KEYWORDS

Software Product Lines, Product Line Evolution, Configurable Systems, Safe Evolution, Partially Safe Evolution, Empirical Study

VAMOS '19, February 6–8, 2019, Leuven, Belgium

ACM ISBN 978-1-4503-6648-9/19/02…$15.00
https://doi.org/10.1145/3302333.3302346

ACM Reference Format:
Karine Gomes, Leopoldo Teixeira, Thayonara Alves, Márcio Ribeiro, and Rohit Gheyi. 2019. Characterizing safe and partially safe evolution scenarios in product lines: An Empirical Study. In *13th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS '19), February 6–8, 2019, Leuven, Belgium.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3302333.3302346

1 INTRODUCTION

Software Product Line (SPL) engineering allows developers to generate customized products according to customers needs in a systematic way through reuse [20]. As with regular software systems, SPLs evolve over time. Adding new features, improving the quality of existing products, or fixing bugs are common changes to SPLs. Nonetheless, in this context, evolution presents different challenges. For instance, changing a single feature might affect a range of valid products. Due to these error-prone changes, research studies focus on understanding SPL evolution aiming to help developers minimize the impact yielded by changes [5, 14, 16, 18, 22].

To support developers on SPL evolution, a refinement theory [5] was proposed, formalizing the *Safe Evolution* notion, which concerns evolution scenarios where the behavior of all existing products should be preserved. Changes that are considered safe by this notion include adding a new optional feature or refactoring some existing asset. Nonetheless, it is often the case that developers indeed desire to change the behavior of existing products. Feature removals, bug fixing, or functional changes, are examples of changes that may affect the behavior of existing products, and thus would be considered unsafe. To consider such changes, Sampaio et al. proposed *Partially Safe Evolution* [22], also based on the refinement theory. This notion concerns evolution scenarios where the behavior of only a subset of the existing products should be preserved.

The theories that establish safe and partially safe evolution notions allow the derivation of transformation templates [3, 5, 17, 22], that abstract a common evolution task, such as adding an optional feature. These templates 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 [3, 17] or partially safe evolution [22] scenarios, and do not examine the

interplay between those two notions. This work aims to characterize product line evolution as a whole, measuring to which extent the evolution history is safe compared to partially safe. Thus, our goal is 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 performed an empirical study that analyzed 2,300 commits from the Soletta Project[1], an open-source framework for Internet of Things applications. We automatically classify commits (evolution scenarios) into templates, which can be done successfully for 78.3% of the commits. As expected, not all commits are automatically mapped to existing templates. Among the 500 remaining commits, 24.4% only modify the mapping between features and code, without changes to the feature model or code. Furthermore, we also classify changes over the remaining commits as safe or partially safe, while also categorizing them using tags, in a way that might help revealing novel templates. Our results show that 91.3% of all 2,300 commits in Soletta are categorized as partially safe evolution scenarios.

In summary, this paper provides the following contributions:

- an empirical study to better characterize SPL evolution, measuring to which extent evolution is safe compared to partially safe;
- a methodology for manually analyzing changes that might reveal novel templates for expressing evolution scenarios.

The remainder of the paper is organized as follows: In Section 2 we explain basic concepts about SPL and its structure, aiming to support the understanding about our study. Section 3 presents our empirical study of Soletta, and Section 4 shows our results. In Sections 6 and 7 we present threats to validity and related work, respectively. Finally, we conclude our study in Section 8.

2 BACKGROUND

In this section, we present an overview of the essential concepts used throughout this work. An SPL is usually organized into three high-level spaces, which we refer to as *Feature Model* (FM), *Configuration Knowledge* (CK), and *Asset Mapping* [5, 19] (AM).

Features are used to specify and communicate the commonalities and differences of the products [2]. They are usually organized into *Feature Models* establishing common and variable features. Such models establish which products can be derived through hierarchy, dependencies, and constraints. Kconfig is a language used to manage variability in the Linux kernel[2] and other SPLs. Listing 1 shows a Kconfig snippe from the Soletta project.

The **config** keyword defines a new named feature. This feature is defined as *boolean*, so it might be optionally selected in a product. The string that follows **bool** is the user-friendly name that might appear in a UI for configuring such models. A **depends on** clause expresses the features dependencies that must be satisfied, while **select** forces the selection of another feature. Finally, **default** establishes the default value for the feature.

Listing 1: Feature declaration in Kconfig

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

Assets make these features concrete and can be of various forms, such as source code, documentation, and images. Listing 2 shows the contents of the `echo-server.c` file. The *Asset Mapping* consists of a mapping of names to the actual assets that might be used.

Listing 2: echo-server.c asset

```
#include "sol-vector.h"
#include "sol-util.h"
struct queue_item {
    struct sol_buffer buf;
    struct sol_network_link_addr addr;
};
```

We then need to associate features to assets (*Configuration Knowledge*). In Kconfig-based systems, we do so through *makefiles*, configuration files written using the `make` language.[3] Listing 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 3: Mapping feature to asset in Makefile

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

2.1 Safe 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 [12, 14, 18]. The concept of *Safe Evolution* [5, 17] 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. A refinement theory formalizes this concept [5], allowing the derivation of transformation templates [3, 17, 23].

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 safe evolution. For instance, Figure 1 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

[1] https://github.com/solettaproject
[2] https://www.kernel. org/doc/Documentation/kbuild/kconfig-language.txt
[3] http://www.gnu.org/software/make/manual/make.html

(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. The ⊑ symbol represents the *refinement* notation, and 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.

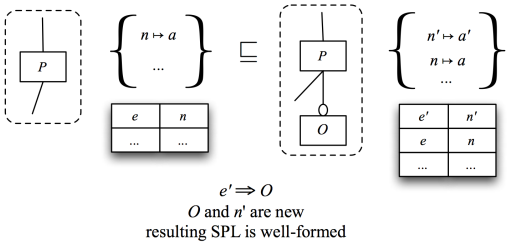

Figure 1: Add New Optional Feature template.

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 [22], 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.

We can also derive transformation templates to abstract partially safe evolution scenarios. For instance, Figure 2 represents the Remove Feature template. 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*.

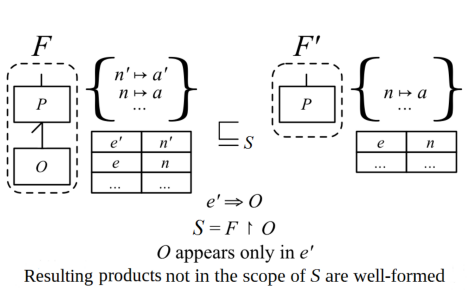

Figure 2: Remove Feature template.

3 EMPIRICAL STUDY

Existing works only focus on evaluating occurrences of either safe or partially safe templates, 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. Thus, our goal is to characterize SPL evolution as a whole, measuring to what extent the evolution history is safe compared to partially safe. Our study aims to answer the following 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?

To answer **RQ1**, we mined 2,300 commits from an existing SPL (each commit is considered as an evolution scenario), classifying them into *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 [11]. 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. We present details over the analyzed SPL in Section 3.1, the automated data extraction process in Section 3.2, and the manual classification in Section 3.3.

3.1 Sample

As our object of study, we used 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 as Linux, using Kconfig to manage variability, C as the main programming language, and Makefiles to map features to code. It is also smaller than Linux, which makes it amenable to manual analysis. We analyze 2,300 commits, ranging from the beginning of the project (June/2015) into the first release (April/2016), to understand the evolution during this lifecycle.

3.2 Methodology

Figure 3 presents the methodology used in our study. In the LHS we illustrate how we extracted information from the repository, while in the RHS 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) [8], Neo4j,[4] and Repodriller.[5]

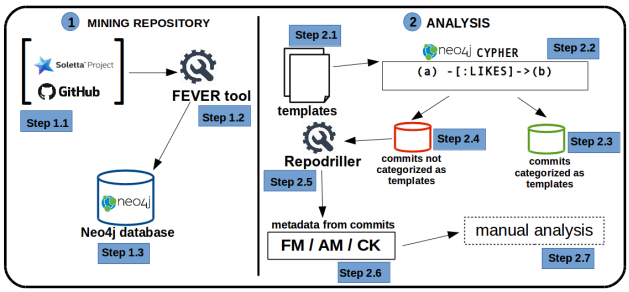

Figure 3: Overview of the methodology for the study.

FEVER (Step 1.2) is a tool for mining Git repositories in a feature-oriented way. The tool extracts detailed information about changes in three spaces: variability model, mapping, and implementation. FEVER output is stored in a Neo4j dataset (Step 1.3). This graph-based dataset contains nodes (*entities*), edges (*relations*), and properties (*attributes*) for each extracted commit. The three SPL spaces are represented as `FeatureEdit` (FM), `MappingEdit` (CK), and `SourceEdit` (AM) entities, respectively. Alternatively, the spaces can also be expressed as `ArtefactEdit` entity with an attribute `type` ranging among `vm`, `build` or `source`.

The data is stored on a Neo4j database. Previous works suggested templates to avoid errors during SPL development (Step 2.1 in Figure 3). Aiming to investigate how often these templates occur in practical scenarios, we use the *Cypher* query language to encode nine existing templates (Step 2.2). Four of those are safe evolution templates: Add New Optional Feature, Add Any Feature without Changing CK and AM, Remove Unused Assets, and Add Unused Assets. The remaining five are partially safe: Remove Assets and CK Mapping, Add Assets and CK mapping, Change Asset, Change CK Lines, and Remove Feature.

To illustrate encoding templates into queries, consider the Add New Optional Feature template from Figure 1. This template expresses an evolution scenario in which a new optional feature is added. We create queries based on the changes described by the template, and the information stored by FEVER in the graph database. Figure 4 shows an example of a query developed in our study to capture evolution scenarios described by the Add New Optional Feature template. This query yields all commits that include a `FeatureEdit` where a feature is added (line 2) as an optional feature (line 3). This commit also includes a `MappingEdit` entity stating that the makefile includes a new mapping associating the new feature name with some asset (line 4). Finally, the query specifies that source files cannot be modified or removed, only additions are allowed, to comply with what the template specifies. We reuse existing queries for the partially safe templates [22].

To check that templates were precisely encoded, we manually checked the results for all queries, to assess their precision. Furthermore, aiming to mitigate the bias of only the first author developing

[4]https://neo4j.com
[5]https://github.com/mauricioaniche/repodriller

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

Figure 4: Query related to Add New Optional Feature template.

and confirming the accuracy of the queries, the third author 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, the second author acted to reach consensus. The protocol for manually reviewing the commits classified by the queries is available in our online appendix [10].

3.3 Classifying the Remaining Commits

Some commits are not matched by any query. So, we perform a semi-automated analysis over such remaining commits. We developed scripts in Repodriller (Step 2.5) to automatically extract certain information from commits, such as what kind of files were changed, from which SPL space, besides general information such as lines added and removed, among others (Step 2.6).

Moreover, we also manually analyzed (Step 2.7 in Figure 3) such commits, categorizing the changes using *tags*. The rationale is that using such strategy, we might reveal opportunities for deriving new templates, if we have sets of tags that frequently appear. We define two dimensions for classifying the changes over these commits: *change type* and *tags*. The *change type* varies among:

- **New:** used 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;
- **Change:** if there is some change in an existent *tag change*;
- **Move:** used 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).

Further, we define *tags* to categorize changes according to the characteristics of each space (Step 2.7). We use the following tags:

- **AM:** `include` (changes on `include` directives); `ifdef` (changes on `ifdef` directives); `changeAsset` (changes that modify an asset); `addAsset` (changes that add an asset); `removeAsset` (changes that remove some asset).
- **CK:** `ifdef` (changes on `ifdef`, `ifneq`, or `ifeq` directives); `mapping` (changes on mappings from features to assets); `build` (rules specifying how assets must be built).
- **FM:** `depends` (changes to `depends` on clause); `select` (changes on `select` expression); `feature` (changes on `config` expression); `default` (changes on `default` expression); `menu` (changes on `menu` expression).

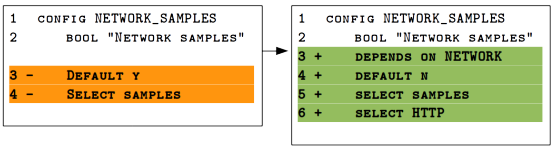

Figure 5: Example of evolution scenario.

We classified each of the remaining commits with these tags and change types, besides categorizing them as safe or partially safe. As an example of applying such tags, suppose the evolution scenario of a Kconfig file illustrated in Figure 5. Before the change (LHS), there was **no instance** of `depends on` clause, and existing `select` and `default` expressions. We should then classify this commit as follows: `new` as *change type* and `depends` as FM *tag*; `change` as *change type* and `default` as FM *tag*; `add` as *change type* and `select` as FM *tag*. 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 mark the commit as *partially safe*. Similar to the manual query analysis, the manual analysis over the remaining commits was also performed by two authors, aiming to ensure consistency over the results and avoid bias. However, in this phase there were no disagreement scenarios.

4 RESULTS

We then present the results of our study described in Section 3.

4.1 Commits classified as templates

After encoding some of the existing 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 our manual analysis, there were doubts over 23 of the commits classified. The second author 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. We ended up with 1,800 commits automatically classified into templates, representing 78.3% of the entire sample. Table 1 shows the amount of commits yielded for each of the templates, after performing the manual analysis of the query precision.

The numbers in Table 1 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. Although the template names might give the intuition that these templates represent solely adding (or removing) an asset, the templates also require that the mapping in CK is also added or removed together with the asset, *with no changes to the FM*. Therefore, we believe this is due because it is more frequent that when assets are added together with changes to the Makefile, it is usually in the context of adding a new feature, which *changes the FM*. Moreover, changes occur in assets already mapped in the Makefile rather than adding or excluding both mapping and asset files together.

Evolution	Template	Commits
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 without change CK and AM	5 (0.21%)
	Remove unused assets	6 (0.26%)
	Add unused assets	34 (1.48%)

Table 1: Amount of commits returned by queries.

There are only two instances of the Remove Feature template. The query actually yields five commits, but there were three false-positives related to renaming, which we manually excluded, as previously mentioned. The Add Unused Assets template had 34 occurrences. Adding an asset without associating it with a feature preserves behavior. These assets might be mapped to some feature later in the evolution history. It might be the case that this later change does not preserve the behavior for some of the products.

Figure 6 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.

4.2 Remaining Commits

This section presents our analysis of the remaining commits, that is, all commits that are not automatically categorized as templates. We first divide the commits into groups according to the SPL space changed and their combination: [AM], [CK], [FM], [FM, CK], [FM, AM], [CK, AM], [FM, CK, AM]. Figure 7 illustrates how each space (and their combination) changes for the remaining commits. We observe that the most frequent group is is that of modifying the CK and AM together [CK, AM], resulting in 137 commits from an amount of 500, representing 27.4% of the remaining commits. The second one is [CK], consisting of 24.4% of the commits. The group with the fewest number of commits is [FM, AM], representing 3%.

We then classify each commit according to the *change type* and *tags* specified in Section 3.3. 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 the Figure 5. 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

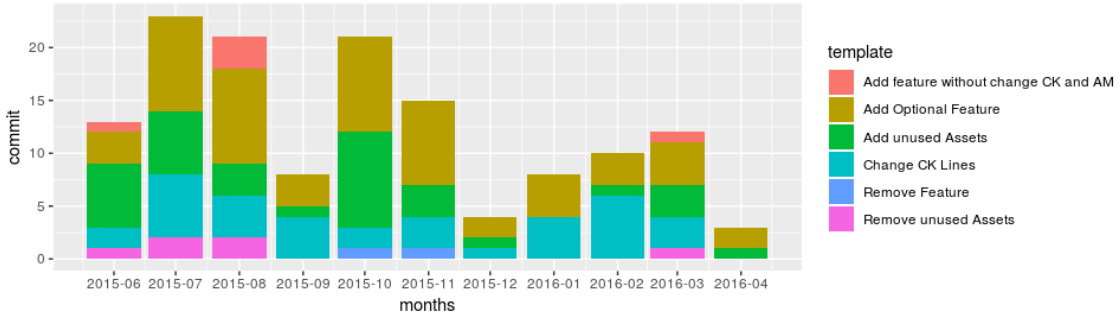

Figure 6: Timeline Templates - *without change asset.*

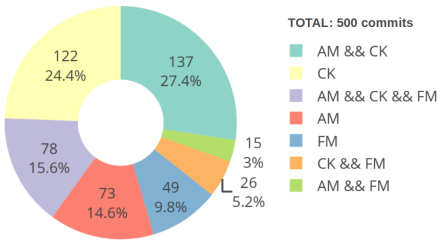

Figure 7: Modified Spaces in Remaining Commits.

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 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 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 code changes (AM) relate 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, 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 mentioned before, there are more *mapping* instances in the AM and CK group than commits which only modify the 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 [5].

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 to 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 scenarios which 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 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, such as .json or .fbp sources, and 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 2 commits remove assets.

5 DISCUSSION

In this section, we discuss our results according to the research questions. 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:

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

Figure 8 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 safer 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 safe evolution.

In terms of patterns for the remaining commits, as aforementioned, commits with changes to both AM and CK 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 scenarios changing only 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.

RQ2: How often templates cover these real scenarios?

As mentioned before, existing templates and their respective queries cover 78% of commits from our 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, CK, and FM 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 semantic conditions, which is an information that cannot be queried against the database.

Despite most evolution scenarios being classified as templates, a reasonable number were still unclassified. In fact, most of existing templates in the catalog [11] focus on changes solely to the FM, or co-evolution of FM and other spaces. However, when evaluating the remaining commits, we observe that some recurrent scenarios can be deeper analyzed to further derive new templates. Our 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).

6 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. [21].

Internal Threat: 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. Although we have tried to mitigate this threat, our manual analysis still might have classified some evolution scenarios incorrectly. Moreover, 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 [10] the study package, including the commits covered by templates and also the dataset with the remaining commits tagged by keywords and change types.

External Threat: 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.

Reliability Threat: Runeson et al. [21] present 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 avaliable in the online appendix, aiming to further reproducibility and replicability of our work.

7 RELATED WORK

The work reported here is based on existing studies over SPL refactoring and evolution [1, 3–5, 16, 22]. Alves et al. [1] 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 [4], illustrating different kinds of refactoring transformation templates that can be useful for deriving and evolving product lines. Borba et al. [5] 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. [16] and Benbassat et al. [3] propose template catalogues to abstract *Safe Evolution* scenarios. Finally, Sampaio et al. [22] extend the refinement theory with the concept of *partial refinement*, establishing the concept of Partially Safe Evolution. This concept allows supporting changes that preserve the behavior for a subset of the existing products. This work differs from these previous studies by focusing on both safe and partially safe evolution templates, to understand the distribution of such changes throughout the SPL evolution history.

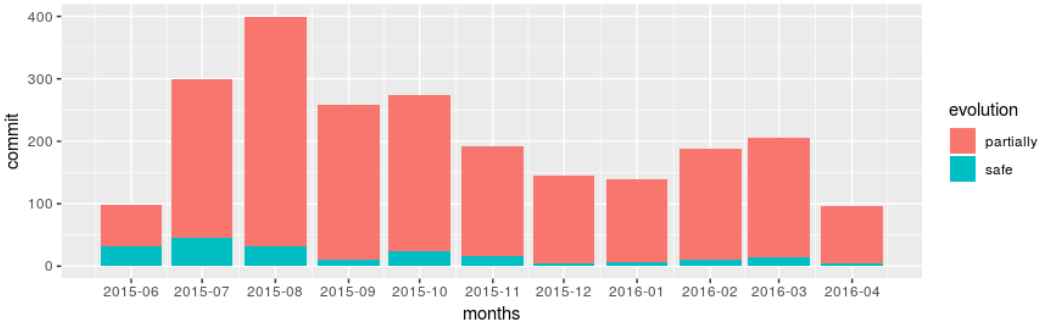

Figure 8: Timeline Safe vs Partially Safe Evolution from 2,300 Soletta commits.

Montalvillo et al. [15] perform a mapping study 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. [7] 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. [9] 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. [19] 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. [6] 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. [13] 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.

8 CONCLUSIONS AND FUTURE WORK

In this work we evaluated 2,300 commits from the evolution history of Soletta, aiming to characterize changes as being *safe* or *partially safe* and matching changes to existing SPL transformation templates. As a result of our analysis, we obtain that **78.3%** (1,800 occurrences) of commits are covered by templates. We verify that in these cases, most of the occurrences refer to the Change Asset template.

For the remaining commits, which amount to 500 occurrences (**21.7%**), we conclude that commits which modify both the CK and AM occur more frequently, explaining around **27.4%** of the commits. Commits that only change the CK are the second most common occurrence instance (**24.4%**). In contrast, commits which present changes in both FM and AM have the fewest instances (3%).

From the remaining commits, evolution scenarios which modify the Kconfig present several changes related to `depends on` clauses. Commits with changes only to the CK present most of the changes related to build rules rather than the mapping between features and assets.

As future work, we intend to perform a deeper analysis over the 1,662 occurrences of the `Change Asset` template. We also intend to tackle the challenge of classifying evolution scenarios that are spread in more than one commit. Moreover, we intend to explore our classification of change types, tags, and modified spaces to derive new templates, for instance supporting co-evolution of FM and CK. Finally, we intend to provide tool-support for SPL developers.

ACKNOWLEDGMENTS

This research was partially funded by INES 2.0,[6] FACEPE grants PRONEX APQ-0388-1.03/14 and APQ-0399-1.03/17, and CNPq grant 465614/2014-0. We also acknowledge support from FACEPE (APQ-0570-1.03/14 and IBPG-0128-1.03/16), CNPq (409335/2016-9, 308380/2016-9, 477943/2013-6, 460883/2014-3, 465614/2014-0, 306610/2013-2, 307190/2015-3, 409335/2016-9, and 426005/2018-0), FAPEAL (PPG 14/2016), and CAPES (175956 and 117875).

[6] http://www.ines.org.br

REFERENCES

[1] Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos Lucena. [n. d.]. Refactoring Product Lines.

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2016. *Feature-Oriented Software Product Lines: Concepts and Implementation* (1st ed.). Springer Publishing Company, Incorporated.

[3] Fernando Benbassat, Paulo Borba, and Leopoldo Teixeira. 2016. Safe Evolution of Software Product Lines: Feature Extraction Scenarios. In *X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*. 11–20. https://doi.org/10.1109/SBCARS.2016.21

[4] Paulo Borba. 2011. 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 (GTTSE'09)*. Springer-Verlag, Berlin, Heidelberg, 1–26. http://dl.acm.org/citation.cfm?id=1949925.1949927

[5] Paulo Borba, Leopoldo Teixeira, and Rohit Gheyi. 2012. A theory of software product line refinement. *Theoretical Computer Science* 455 (2012), 2 – 30. https://doi.org/10.1016/j.tcs.2012.01.031

[6] Johannes Bürdek, Timo Kehrer, Malte Lochau, Dennis Reuling, Udo Kelter, and Andy Schürr. 2016. Reasoning about product-line evolution using complex feature model differences. *Automated Software Engineering* 23, 4 (01 Dec 2016), 687–733. https://doi.org/10.1007/s10515-015-0185-3

[7] Nicolas Dintzner, Arie Van Deursen, and Martin Pinzger. 2013. Extracting Feature Model Changes from the Linux Kernel Using FMDiff. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS '14)*.

[8] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. 2016. FEVER: Extracting Feature-oriented Changes from Commits. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/2901739.2901755

[9] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. 2018. FEVER: An approach to analyze feature-oriented changes and artefact co-evolution in highly configurable systems. *Empirical Software Engineering* 23, 2 (01 Apr 2018), 905–952. https://doi.org/10.1007/s10664-017-9557-6

[10] Karine Gomes. 2018 (accessed november, 2018). *Vamos 2019 - A Case Study to Characterize Evolution in Software Product Line*. http://www.cin.ufpe.br/~kgmg/vamos2019/.

[11] SPG Group. 2018 (accessed november, 2018). *Templates for Software Product Line Evolution*. https://github.com/spgroup/pl-refinement-templates-catalog/blob/master/templatescatalog.pdf.

[12] W. Heider, M. Vierhauser, D. Lettner, and P. Grünbacher. 2012. A Case Study on the Evolution of a Component-based Product Line. In *Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*. 1–10. https://doi.org/10.1109/WICSA-ECSA.212.8

[13] Christian Kröher, Lea Gerling, and Klaus Schmid. 2018. 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 (SPLC '18)*. ACM, New York, NY, USA, 54–64. https://doi.org/10.1145/3233027.3233032

[14] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. 2010. Evolution of the Linux Kernel Variability Model. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond (SPLC'10)*. Springer-Verlag, Berlin, Heidelberg, 136–150. http://dl.acm.org/citation.cfm?id=1885639.1885653

[15] Leticia Montalvillo and Oscar Díaz. 2016. Requirement-driven evolution in software product lines: A systematic mapping study. *Journal of Systems and Software* 122 (2016), 110 – 143. https://doi.org/10.1016/j.jss.2016.08.053

[16] Laís Neves, Paulo Borba, Vander Alves, Lucinéia Turnes, Leopoldo Teixeira, Demostenes Sena, and Uirá Kulesza. 2015. Safe evolution templates for software product lines. *Journal of Systems and Software* 106 (2015), 42 – 58. https://doi.org/10.1016/j.jss.2015.04.024

[17] Laís Neves, Paulo Borba, Vander Alves, Lucinéia Turnes, Leopoldo Teixeira, Demostenes Sena, and Uirá Kulesza. 2015. Safe Evolution Templates for Software Product Lines. *Journal System Software* 106, C (Aug. 2015), 42–58. https://doi.org/10.1016/j.jss.2015.04.024

[18] Leonardo Passos, Krzysztof Czarnecki, and Andrzej Wasowski. 2012. Towards a Catalog of Variability Evolution Patterns: The Linux Kernel Case. In *Proceedings of the 4th International Workshop on Feature-Oriented Software Development (FOSD '12)*. ACM, New York, NY, USA, 62–69. https://doi.org/10.1145/2377816.2377825

[19] Leonardo Passos, Leopoldo Teixeira, Nicolas Dintzner, Sven Apel, Andrzej Wasowski, Krzysztof Czarnecki, Paulo Borba, and Jianmei Guo. 2016. Coevolution of variability models and related software artifacts. *Empirical Software Engineering* 21, 4 (01 Aug 2016), 1744–1793. https://doi.org/10.1007/s10664-015-9364-x

[20] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[21] Per Runeson, Martin Host, Austen Rainer, and Bjorn Regnell. 2012. *Case Study Research in Software Engineering: Guidelines and Examples* (1st ed.). Wiley Publishing.

[22] Gabriela Sampaio, Paulo Borba, and Leopoldo Teixeira. 2016. Partially Safe Evolution of Software Product Lines. In *Proceedings of the 20th International Systems and Software Product Line Conference (SPLC '16)*. ACM, New York, NY, USA, 124–133. https://doi.org/10.1145/2934466.2934482

[23] Leopoldo Teixeira, Vander Alves, Paulo Borba, and Rohit Gheyi. 2015. A Product Line of Theories for Reasoning About Safe Evolution of Product Lines. In *Proceedings of the 19th International Conference on Software Product Line (SPLC '15)*. ACM, New York, NY, USA, 161–170. https://doi.org/10.1145/2791060.2791105