On the Evolution of Feature Dependencies: An Exploratory Study of Preprocessor-based Systems

Raiza Oliveira, Bruno Cafeo, Andre Hora
Faculty of Computer Science
Federal University of Mato Grosso do Sul
Campo Grande, Brazil
raiza.a.oliveira@aluno.ufms.br,{cafeo,hora}@facom.ufms.br

ABSTRACT

Feature dependencies are the subject of several studies and have been approached with different goals, such as software quality. However, we still It is not known whether feature dependencies are prone to change during the lifetime of configurable systems. To minimize this lack, we perform an empirical study to assess the evolution of feature dependencies. We focus in feature dependencies at the source-code level written with preprocessor directives, for this, we analyze the history of feature dependency changes in 15 preprocessor based systems written in C. In order to identify the most common types of changes on feature dependencies, we propose a catalog of possible changes in these dependencies. We found out that about 9% of the feature dependencies between two subsequent versions are added, and less of 4% are excluded. Whereas, about 87% of the dependencies between two subsequent versions are preserved, and 14% has some kind of structural modification. Besides, the results show that functions are 3x more prone to change than variables in feature dependencies. In summary, our results show that: (*i*) feature dependencies prone to be preserved during the evolution; (*ii*) about 1/5 of the dependencies preserved between two versions have structural changes; (*iii*) functions changes more than variables; (*iv*) developers may have to deal with 1/4 of the dependencies on each evolution of the configurable system.

CCS CONCEPTS

• **Software and its engineering → Software evolution**; **Maintaining software**; *Software maintenance tools*; Application specific development environments;

KEYWORDS

Configurable Systems, Feature Dependencies, Preprocessor, Maintenance

ACM Reference Format:
Raiza Oliveira, Bruno Cafeo, Andre Hora. 2019. On the Evolution of Feature Dependencies: An Exploratory Study of Preprocessor-based Systems . 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/10.1145/3302333.3302342

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

ACM ISBN 978-1-4503-6648-9/19/02…$15.00
https://doi.org/10.1145/3302333.3302342

1 INTRODUCTION

Software has become configurable to tailor development of individual products for a wide range of market segments [2]. Configuration options, also known as features [14], range from options that enable small functional aspects to support alternative hardware, provide backward compatibility, add debugging facilities, to those that implement whole subsystems [20].

Features in configurable systems relate to each others to fulfill specific tasks [6]. In the source code, a feature dependency occurs whenever one or more program element within the boundaries of a feature depend on elements external to that feature, such as a method defined in one feature and called by another feature [29].

Lately, several studies point out the importance of feature dependencies in the context of maintenance of configurable systems. For instance, previous studies report that it is reasonably common to find feature dependency implementation along the source code of configurable systems [3, 4, 29]. The research community also began discussing the impact of feature dependency as one of the indicators of quality in configurable systems [6–8, 12, 29, 31]. In addition, researchers propose approaches and tools to support the maintenance of configurable system code in the presence of feature dependencies [9, 28, 32].

Despite the key role played by feature dependencies in the maintenance of configurable systems, little is known about whether and how feature dependencies evolve in the context of the evolution of configurable systems. It is crucial to provide developers with knowledge about how feature dependencies evolve so that changes in configurable systems can be better planned and the maintenance effort minimized.

Therefore, to minimize this lack and better understand how feature dependencies evolve, we perform an exploratory study to assess the evolution of feature dependencies throughout the evolution of configurable systems. In particular, we want to unveil (i) the frequency of additions, exclusions and modifications of feature dependencies, and in case of modifications (ii) explore the changes in program elements participating of the implementation of feature dependencies at source-code level. To achieve our goals, we conducted an exploratory study comprising a total number of 918 releases and more than 141M lines of code on the evolution of 15 open source configurable systems.

In summary, we shed light on the following findings:

- About 73% of the dependencies are not modified during the evolution of configurable systems.
- Despite the majority of preserved dependencies, we find a considerable number of changes in the implementation of those dependencies, the so-called fine-grained changes (approx. 14%). Summed up with additions and deletions,

approximately 26% of the dependencies are different between subsequent versions.
- Functions are 3x more prone to change than variables when participating in feature dependencies implementation. Furthermore, changes in parameter list and return type comprises more than 50%. These types of changes together with others listed in our proposed catalog indicates a high probability of impact on dependent features when a function is changed in the context of the implementation of a feature dependency.

The remainder of this paper is structured as follows. Section 2 presents background on the main topics of this paper as well as a motivating example of the target problem. The catalog of changes in feature dependencies is described in Section 3 A complete description of our study is provided in Section 4. Section 5 presents the data collected during the exploratory study and an analysis of the results. Section 6 presents related work. Finally, Section 7 discusses the limitation of this work and Section 8 concludes the paper with some remarks and future directions.

2 PRELIMINARIES

To lay a foundation for subsequent sections, we introduce the basic concepts of configurable systems and feature dependency. We also present a motivating example to illustrate the evolution of a feature dependency implementation.

2.1 Feature dependency

Systems that share a common core but also have different functionalities are referred to as configurable systems [2]. The core implements the basic functionality present in any member of a program family, and the different selections of configurations define the set of program variants. These commonalities and variabilities are often modeled as features, each representing additional functionality to the core software [26].

When we consider configurable systems written in C, developers often use the C preprocessor to annotate the implementation code of features. The preprocessor identifies the code that should be compiled or not based on preprocessor directives (i.e. `#ifdef` directives) along with a macro expression. Macro expressions might be composed by one or more macros, as a boolean formula, which refers to specific features. Developers use preprocessor directives to wrap from entire structures of code such as functions to a single variable. In this context, we consider a feature is a set of program elements (functions or variables) surrounded by preprocessor directives.

A feature dependency defines how the implementation of a feature is dependent on the implementation of other features. A feature dependency occurs in the source code of a configurable system whenever one or more program elements (e.g. code blocks, functions or variables) within the boundaries of a feature depend on elements external to that feature [6]. In other words, feature dependencies are established by means of structural dependencies in the source code between elements of different features [6]. Such as function declared in one feature e called in another feature.

To better explain this concept, we refer to the code snippet in Figure 1 extracted from release 1.4.39 of Lighttpd.[1]

[1] https://github.com/lighttpd/lighttpd1.4

```
1  #ifdef P2MP_SERVER
2  . . .
3  bool hash_add (struct hash *hash, const void
4  *key,
5    void *value){
6    uint32_t hv;
7    . . .
8    bool ret = false;
9    hv = hash_value (hash, key);
10 . . .
11   return ret;
12 }
13 . . .
14 . . .
15 #ifdef MANAGEMENT_DEF_AUTH
16   do {
17     . . .
18   } while (!hash_add (m->cid_hash,
19   &mi->context.c2.mda_context.cid, mi));
20   . . .
21 #endif
```

Figure 1: Program element configuring a feature dependency.

The code snippet of Figure 1 presents parts of the source code of two features: P2MP_SERVER and MANAGEMENT_DEF_AUTH. In this case, feature MANAGEMENT_DEF_AUTH depends on P2MP_SERVER by means of the function `hash_add`. Lines 18-19 from feature MANAGEMENT_DEF_AUTH call the function `hash_add` which belongs to feature P2MP_SERVER.

It is very important to notice that the dependency between these two features in Lighttpd is made by other program elements further than `hash_add`. In fact, a feature dependency is often composed by several program elements [9].

2.2 Motivating example

Changes in the source code are expected when configurable systems evolve. As a consequence the set of feature dependencies in a system can also evolve. Feature dependencies can be (i) added, (ii) excluded or (iii) modified. A feature dependency is added when one or more program elements within the boundaries of a feature begins to depend on elements external to that feature. Similarly, a feature dependency is excluded when every reference to external program elements are excluded from the dependent feature. Finally, and more complicated for developers to recognize, changes can touch the source code related to the implementation of a feature dependency changing its structure. In other words, one or more program elements participating of the implementation of a feature dependency can be changed during the system's evolution.

Figure 2 illustrates a possible change in one program element participating of the implementation of a feature dependency (i.e., feature dependency modification). Notice that this dependency may have other program elements participating of its implementation. Figure 2(a) (Lines 3-4) presents the function `hash_add` which is part of the implementation of a feature dependency (Section 2.1). Figure 2(b) (Lines 3-4) shows a change in such program element in a subsequent version of the source code. A new parameter was added in the function `hash_add`.

Several changes can happen to modified feature dependencies. To cite a few: addition or exclusion of references to program elements, change of the return type of a function, change of the specifier of a variable and change of the parameter list of a function.

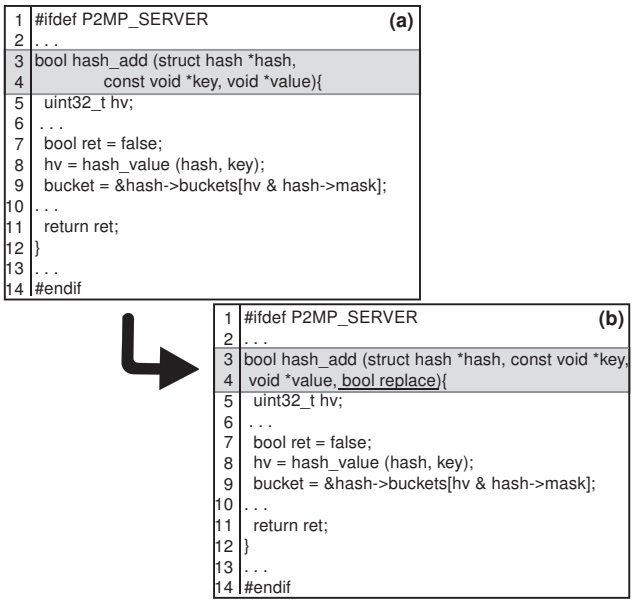

Figure 2: Function parameter insertion.

The problem of evolving feature dependencies is that as a single feature is often not localized in a single module in the implementation of configurable systems [8, 15, 27], developers may not fully recognize additions, exclusions and modifications of feature dependencies throughout the evolution of configurable systems.

So, as a first step to systematically understand the evolution of feature dependencies, the next section presents a catalog of possible changes in program elements in the context of a feature dependency. When a feature dependency is modified, at least one of those changes should happen. In other words, this catalog contains every type of change that may happen on the implementation (i.e., program elements) of existing dependencies during the evolution of configurable systems as shown in Figure 2. The types of changes described in the catalog encompass source code written in C and using preprocessor directives to implement features.

3 CATALOG OF FEATURE DEPENDENCY CHANGES

This section describes the possible changes that can happen with program elements implementing feature dependencies in configurable systems implemented in C. The intent of this catalog is to allow developers and researchers to understand and classify the changes that can occur in program elements that may be part of the implementation of feature dependencies, provide a basis for discussion towards the modification of feature dependencies as well as their impacts on maintenance, and give a representative list of changes that may be used for developers to implement tools in order to support the evolution of feature dependencies.

To build the catalog of changes in program elements participating of the implementation of feature dependencies, we first studied the C language syntax in order to identify how feature dependencies are usually implemented in preprocessed configurable systems. Second, we manually analyzed several open-source configurable systems in order to identify changes on those program elements and start our classification. Third, we refer to related work that deals with either changes in preprocessed systems [19, 25] or patterns of changes [5, 10, 34] (e.g. changes in APIs) to better organize our catalog.

The catalog presents program elements that can participate in the implementation of feature dependencies and properties of those program elements that can be changed. As explained in Section 2.2, a change in a feature dependency usually occurs in properties of one or more program element participating of the implementation of a dependency. According to the definition of feature dependency presented in Section 2.1, the program elements considered in this catalog are: (i) functions and (ii) variables. Each program element is composed by a combination of properties. In the following, we list the properties provided by the C language to the program elements considered in this catalog.

Type (T). It is used to indicate how much memory should be dedicated to store data. C language has five primitive types: `void`, `char`, `int`, `float` and `double`.

Modifier (M). It changes the size allocated to a element. As a rule, they are associated to primitives types. Except for the `void` type, all the others types may have modifiers. Examples of modifiers are `signed`, `unsigned`, `short` and `long`.

Specifier (S). It determines whether the element has a global or local lifetime. The specifier can be `extern`, `auto`, `static` or `register`.

Qualifier (Q). It is a way of expressing additional information about a value through the type system, and ensuring correctness in the use of the data. The qualifiers provided by C language are `volatile` or `const`. For example, when a variable has a `const` qualifier associated, one say that this variable is a constant.

In the following, we present the catalog of feature dependency changes (Table 1). For each program element, we present a list of possible changes in its properties.

Table 1: Catalog of changes in program elements implementing feature dependencies.

Program Element	List of Changes
Variable	Inclusion of variable, exclusion of variable, changes of: type, modifier, qualifier and/or specifier
Function	Inclusion of function, exclusion of function, change of: return type, modifier, qualifier, specifier and/or parameter list

Next, we explain the structure to define each program element in C considered in our catalog. We also provide example of a possible change that may happen during the evolution of the feature dependency code for each program element in the catalog.

It is important to mention that it is out of the scope of this paper to analyze the impact of feature-dependency evolution in the source code. In other words, we are not concerned about the impact of adding, excluding or modifying feature dependencies in the source code of configurable systems such as compilation errors, change propagation, and so on.

3.1 Variable

Variables are names used to refer to some location in memory that holds a value. The declaration of a variable should have the following syntax:

$$[Q]\ [S]\ [M]\ T < variable\ name >;$$

The definition of qualifiers (Q), specifiers (S) and modifiers (M) are optional when declaring variables. However, the variable type (T) is always mandatory as well as the variable name. Figure 3 shows a change associated to a variable participating of the implementation of a feature dependency. This change happened in an evolution of GNUPlot.

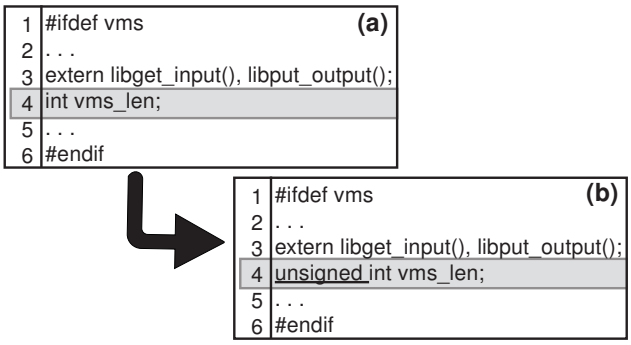

Figure 3: Variable modifier change.

Variable `vms_len` (Line 4 - Figure 3.A) is declared with no explicit definition of qualifier, specifier and modifier in a specific release. Note that those properties have default values even when developers do not explicit define them. In the subsequent release, there is a explicit definition of the variable modifier to `unsigned` (Line 4 - Figure 3B). This change may cause problems if a dependent feature using `vms_len` expects a signed value.

3.2 Function

A function is a group of statements that together perform a task. The definition of a function has the following structure:

$$[Q]\ [S]\ [M]\ T < function\ name >\ ([< parameter\ list >])\{\ldots\}$$

Explicit defining qualifiers (Q), specifiers (S), modifiers (M) and the parameter list are optional when defining functions. However, the return type (T) and the function name are always mandatory. Figure 4 shows a change associated to a function participating of a feature dependency. This change happened in an evolution of OpenSC[2] project.

[2]https://github.com/OpenSC/OpenSC

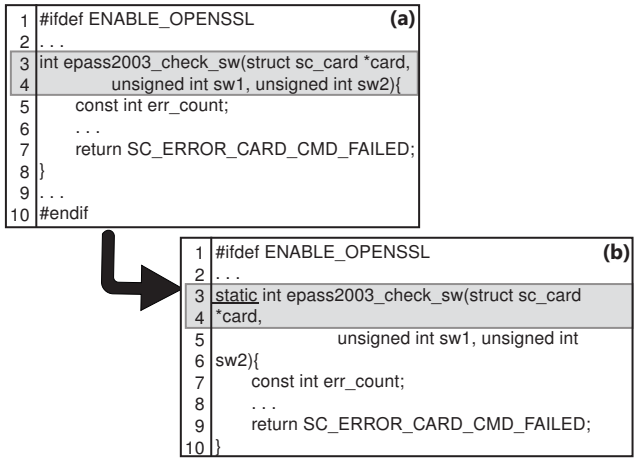

Figure 4: Function return type change.

Function `epass2003_check_sw` (Line 3 - Figure 4.A) is declared with no explicit definition of qualifier, specifier and modifier in a specific release. The default value of a function specifier is `auto` when it is not explicit defined. In the subsequent release, there is a explicit definition of the function specifier to `static` (Line 3 - Figure 3.B).

4 RESEARCH SETTING

In this section, we describe our study in terms of its goals (Section 4.1), the subject system used in our study (Section 4.2), and the data collection procedure (Section 4.3).

4.1 Research Questions

During the evolution of a configurable system, it is important to minimize changes in feature dependencies. Adding, excluding or modifying feature dependencies during the evolution increase the maintenance effort [6]. In this context, one may expect that the dependencies of the system remain relatively stable throughout evolution. On the other hand, prior studies point out to several problems related to feature dependencies at source-code level. The results of those studies lead us to believe that feature dependencies evolve together with the configurable system. Therefore, it would be interesting to explore whether and how feature dependencies evolve. Thus, the first research question addresses the number of additions, exclusions and modifications of feature dependencies throughout evolution of configurable systems: [*RQ*1] *How feature dependencies evolve in configurable systems in terms of additions, exclusions and modifications?*

If feature dependencies are often modified, it is important to unveil what exactly changes. The source code of a single feature dependency may be distributed all over the code. This means that several changes may occur in a single dependency during an evolution. In other words, we aim at understanding which changes are happening in each program element of every feature dependency. The goal is to provide a basis for approaches and tools to support developers. To do so, we use our catalog of changes in program elements (Section 3) to assess which are the most common changes in elements of feature dependencies. Thus, the second research question is concerned about what changes

happen in feature dependencies: [*RQ2*] *How do feature dependency implementations change throughout the evolution of configurable systems?*

4.2 Subject Systems

We analyze 918 different releases of 15 configurable systems written in C with preprocessor directives, totalizing more than 141 million lines of code. For each system, we analyze the whole history available in the repository in terms of releases. Our selection covers different domains, such as file compression software and text editor. We select these subject systems inspired by previous work [18, 24, 25, 31]. To better characterize these systems, Table 2 presents the configurable systems used, number of releases analyzed, age of the system based on the first and last release, total number of lines of code analyzed considering all releases, total number of features analyzed, average number of feature per release, total number of feature dependencies (FD) analyzed, and average number of feature dependencies per release.

4.3 The Data Collection Procedure

The data collection procedure comprises the following steps: (i) mine software repositories, (ii) identify feature dependencies, and (iii) detect additions, exclusions and modifications in feature dependencies. It is important to mention that the entire data collection was automated. We perform the exploratory study using a 1.6 GHz quad-core Intel Core i5-4200U with 8GB of RAM, running Windows 10 Pro 64-bit. Next we describe each activity of the data collection procedure.

Mining software repositories. The first step of the data collection procedure was to mine the repository of the selected projects in order to analyze the historical changes in feature dependencies. We implemented a tool that retrieve each release of the project identified by tags in the git repository.

Identifying feature dependencies. After retrieving the source code of a specific project release, the next step was to identify feature dependencies. As mentioned in Section 2.1, we detect relationships between program elements (i.e., functions and variables) in different features. This step was implemented in our tool on top of TypeChef [16], a variability-aware type checking utility. TypeChef creates an Abstract Syntax Tree (AST) from each source file and preserves all variability information, having each node in the AST an associated macro of preprocessor directive. Thus, we identified related nodes with different macros associated in order to detect dependencies and their program elements.

Detecting the evolution of dependencies. To detect the evolution of feature dependencie, we stored data of two subsequent releases in order to compare them. In this step, we detected added, excluded, modified and not modified feature dependencies.

5 RESULTS

This section presents the results[3] of the analysis described in Section 4 and perform exploratory analysis of the results. Section 5.1 describes the results of the modifications over feature dependencies during the evolution of configurable systems to answer RQ1. Section 5.2 shows the results and analysis about the data related to fine-grained changes in feature dependencies based on our proposed catalog to answer RQ2.

[3]The results are available at the https://facom.ufms.br/~cafeo/vamos2019/index.html

5.1 The evolution of feature dependencies

As pointed out in Section 4.1, due to the number of related work [6, 12, 28] identifying several maintenance problems related to feature dependencies, it is desirable that feature dependencies do not change frequently throughout evolution of configurable systems. To understand whether feature dependencies change often or not and answer RQ1, we explore the number of feature dependencies added, excluded, modified and not modified in each evolution of all projects. Figure 5 shows the number of added, excluded, modified and not modified feature dependencies throughout evolution of each target system.

As we can notice in Figure 5, the majority of feature dependencies were preserved throughout the evolution of configurable systems. More exactly, an average of 87.14% of the dependencies were preserved. An average of 9.41% of dependencies are new, while 3.45% are deleted during the evolution of the analyzed systems. Based on this data, we shed light to the following analysis. In fact, the number of not modified feature dependencies is statistically significant different from the number of added, excluded and modified feature dependencies ($p\text{-}value < 0.05$ for Mann-Whitney test) with large effect (*effect-size* is -0.6598878 for Cliff's Delta). This means that most of feature dependencies are not modified throughout the evolution of the system.

Despite of the stability of feature dependencies, one can observe a considerable number of dependencies being added, excluded and modified in all target systems. The evolution of these feature dependencies increase the maintenance effort. Therefore, in the following, we present the main findings related to the evolution of those feature dependencies.

Growing number of feature dependencies. We observe the higher number of added dependencies when compared to excluded dependencies. This leads us to believe that developers will always deal with more and more dependencies. Therefore, an increasing effort is expected even when one may imagine a relative stability of the system in the course of time.

Considerable difference between releases. Another important issue is the total number of feature dependencies that are different from one version to the subsequent version. When feature dependencies are added or excluded, the developer needs to revisit the source code in order to comprehend the changes in terms of dependencies and keep the system up-to-date. In other words, the smaller the difference between dependencies from subsequent versions the better. However, the numbers collected are indicating a considerable difference between versions. An average of 1.6 dependency is deleted per evolution per system, whilst 2.1 dependencies are added. This means that 3.7 dependencies out of an average of 22.4 dependencies are different in each evolution. In other words, the developer needs to deal with approximately 16% of the total number of dependencies in each evolution of the system.

Modified dependencies are relevant. The implementation of preserved feature dependencies can be modified throughout the evolution of the systems. This occurs when changes happen in properties of program elements implementing a dependency or when program elements are included or excluded in the

Table 2: Set of subject systems.

System	# of Releases	Age (Years)	LOC (Total)	Features (Total)	Avg. # Features	FD (Total)	Avg. # FD
Curl	177	18	13,457,614	7,220	40.8	5,974	33.7
Emacs	130	33	52,043,700	4,898	37.7	3,495	26.8
Expat	39	20	549,464	190	4.9	155	4.0
GZip	7	25	98,362	56	8.0	39	5.6
Hexchat	17	8	1,329,498	228	13.4	287	16.9
Lighttpd	56	13	4,704,730	1,531	27.3	2,159	38.6
MPSolve	6	5	363,862	63	10.5	24	4.0
OpenVPN	76	11	9,904,685	2,050	27.0	2,215	29.1
OpenSC	27	7	3,230,968	440	16.3	552	20.44
Ossec hids	33	13	3,517,614	1,457	44.2	2,575	78.03
Sylpheed	105	12	11,677,224	1,226	11.7	2,290	21.81
Pacemaker	86	10	8,693,575	554	6.4	330	3.84
Gnuplot	40	29	3,865,111	1,101	27.5	784	19.60
PidGin	86	12	26,488,633	2,663	31.0	2,649	30.80
LibSSH	33	10	1,225,425	957	29.0	1,707	51.73
AVERAGE	**61.2**	**15.1**	**9,410,031**	**24,631**	**22.4**	**25,235**	**25.7**

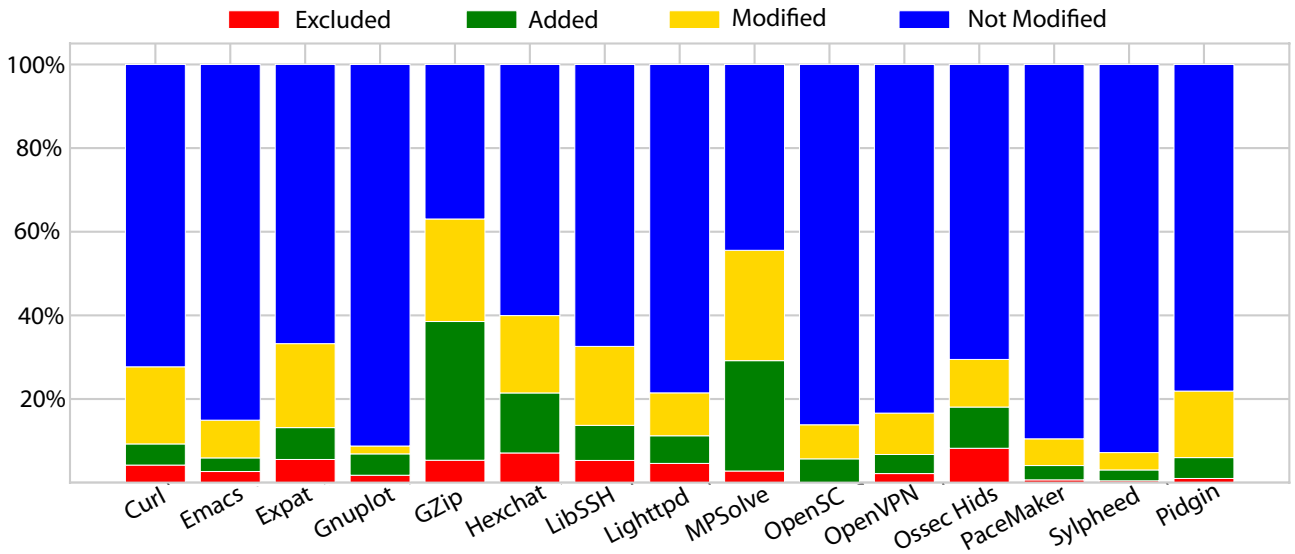

Figure 5: Frequency of modifications in feature dependencies.

composition of a dependency (Section 3). Such "fine-grained changes" are problematic because it is challenging to identify them in the source code. This means that developers may unintentionally change program element participating in the implementation of feature dependencies. In this case, developers may overlook important parts of the code that should be revised or changed due to the change of the program element. Conversely, they may also analyze unnecessarily parts that are not relevant to the evolution, thereby increasing the maintenance effort. When analyzing Figure 5, the number of modified feature dependencies arises as an important issue to explore. The average of feature dependencies preserved that suffer changes is 13.59%. This number of fine-grained changes summed up with the number of additions and deletions gives us approximately 26% of feature dependencies different between subsequent versions. In other words, approximately 14 of the dependencies should be analyzed by the developer in each evolution of the system.

5.2 Modified feature dependencies

Section 5.1 points out the importance of understanding modified feature dependencies. Thus, as a first step towards understanding changes in program elements of modified dependencies, we categorize an explore them in order to observe the most frequent changes.

Composition of dependencies and program element changes. To understand the data collected to answer RQ2, we need to assess the composition of feature dependencies in terms of program elements. Table 3 presents the average number of

program elements (PE) per feature dependency, as well as the distribution of these program elements in terms of variables and functions. We can observe an average of 9 program elements per feature dependency. We also notice an average number of approximately 7 changes in program elements per feature dependency. This means that more than 55% of the feature dependency composition is changed when a dependency is modified. In the context of configurable systems, where the implementation of feature dependencies are scattered in the source code, this high number of change in program elements may increase substantially the maintenance effort depending on the total number of feature dependencies changed (about 18% as shown in Section 5.1).

Table 3: Fine-grained changes in feature dependencies.

Project	# of PE	# of var.	# of func.	Changes per FD
Curl	2.6	2.7	4.3	6.9
Emacs	8.2	4.3	4.0	2.7
Expat	5.4	4.8	3.4	1.0
GZip	9.6	3.8	5.8	1.8
Hexchat	3.9	2.1	4.6	3.4
Lightpd	6.4	2.0	4.4	4.3
MPSolve	4.1	1.3	2.8	1.4
OpenVPN	1.5	0.5	4.5	3.3
OpenSC	6.4	1.2	5.2	2.1
Ossec hids	3.7	0.9	2.8	22.9
Sylpheed	0.8	0.2	0.6	0.9
Pacemaker	3.5	0.8	2.7	0.7
Gnuplot	1.4	0.2	1.2	0.4
PidGin	3.4	0.6	2.8	4.7
LibSSH	4.7	0.6	4.1	18.2
AVERAGE	4.4	1.7	3.5	6.9

Variables vs. Functions. Feature dependencies are composed by variables and functions. Therefore, an important data to analyze is the ratio of changes in terms of variables and functions. Table 3 shows the average number of variables and functions implementing a single feature dependency, and the average number of changes per variable and per function. We have a majority of functions implementing a feature dependency. About 73% of a feature dependency is composed by functions and 27% by variables. Our data also shows a higher number of changes in functions when compared to variables (almost 3x higher). In fact, the number of changes in functions is statistically significant different from the number of changes in variables (*p-value* = 2.2e-16 for Mann-Whitney test) with large effect (*effect-size* is -0.5388003 for Cliff's Delta). Besides the higher number of function when compared to variables, another fact that may explain those numbers are the higher possibilities of changes in functions when compared to variables. Based on our catalog (Section 3), there are seven possible changes in dependencies involving functions whereas there are six possible changes in variables. This fact is explored in the following analysis. Nevertheless, developers should pay extra attention when changing functions participating of a feature dependency.

Changes in variables. To better understand changes in program elements, we explore the changes in the properties of those program elements and the inclusion and exclusion of those elements in the implementation of feature dependencies. Looking at the data related to variables and considering the catalog presented in Table 1, there may be six changes related to variables in the implementation of feature dependencies: addition or exclusion of a variable reference, change in type, modifier, qualifier, and specifier. Figure 6 shows the distribution of changes happened in variables in the context of dependencies. As we can observe, there are categories dominating the changes related to variables. In fact, at least one category stochastically dominates other categories (*p-value* = 0.01102 for Kruskal-Wallis test by rank). Addition and exclusions of variables comprise more than 61% whereas type change involves about 25%. Summing up these three modifications, we have more than 86% of changes involving variables participating in feature dependencies. It is also important to notice that the number of changes between categories is statistically significantly different in all pairwise comparison (*p-value* < 0.05 for Mann-Whitney test for all pairs of categories) except for the pair comprising addition and exclusion. Moreover, there is a large effect size (extracted from Cliff's Delta) for every comparison between each of these three categories (inclusion, exclusion and type change) with the remaining categories. One point that deserves attention is the fact that exclusion of variables and type change are modifications that may demand changes in dependent features. Therefore, there is a considerable high probability that a change in a variable implementing a feature dependency impacts dependent features.

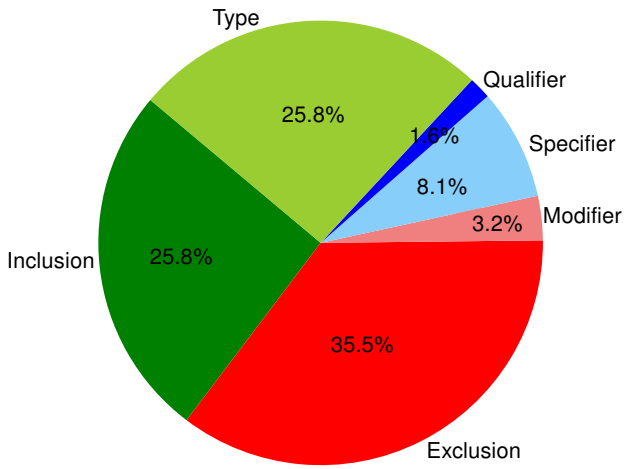

Figure 6: Distribution of changes involving variables.

Changes in functions. Functions are the other program elements that can participate of the implementation of feature dependencies. Based on the catalog presented in Table 1, it is possible to have seven changes related to functions in the implementation of feature dependencies: addition or deletion of a function reference, change in return type, modifier, qualifier, specifier, and parameter list. Figure 7 shows the distribution of changes happened in functions in the context of dependencies. As we can observe, there are categories dominating the changes related to functions. In fact, at least one category stochastically dominates other categories (*p-value* = 0.006941 for Kruskal-Wallis test by rank). The majority of changes involves addition of

functions and changes in parameter list (more than 57%). However, changes touching return type and exclusion of functions references comprises more than 35% of the changes related to functions. It is also important to notice that the number of changes between categories is statistically significantly different in all pairwise comparison ($p\text{-}value < 0.05$ for Mann-Whitney test for all pairs of categories) except for the pair comprising return type and exclusion. Moreover, there is a large effect size (extracted from Cliff's Delta) for every comparison between each of these four categories (inclusion, exclusion, return type and parameter list) with the remaining categories. This may be more problematic than variables because of changes in both parameter list and return type, as well as exclusion of functions (approx. 50%), are likely to demand changes in dependent features. Since functions are changed more than variables, and about 64% of those changes may affect dependent features, developers should pay attention to those changes, and researchers could explore these specific changes to support developers with studies and tools.

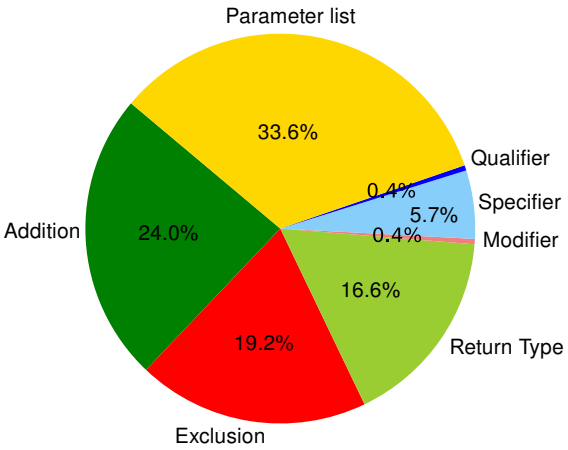

Figure 7: Distribution of changes involving functions.

6 RELATED WORK

FEATURE DEPENDENCIES. Several studies [1, 6, 9, 17, 28–32] approach how feature dependencies occur in practice, identifying their characteristics and frequency. They found a high number of feature dependencies and indicate that the awareness of feature dependencies decreases the effort on maintenance tasks. In addition, some propose tools and approaches to support the maintenance process. Our work focuses in fragments of the feature code, specifically program elements of dependencies in order to understand changes in program elements happened in feature dependencies implementation in the context of evolution, in order to increase the awareness of developers about feature dependencies and how they evolve.

C PREPROCESSOR. Prior studies investigated properties of C preprocessor usage in different contexts and with different goals [11, 13, 18, 19, 21, 23–25, 27, 27]. Those studies analyzed product families implemented in C with the preprocessor to gather information about scattering and tangling of feature code. Our work focuses on changes in preprocessor code. However, we take into account program elements shared across different macros, and we are not concerned about the use of preprocessor directives, but we aim to understand feature dependencies changes throughout configurable systems evolutions.

7 STUDY LIMITATIONS

This section discusses the study limitations based on the four categories of validity threats described by Wohlin et al. [33]. The chosen systems came from different application domains (Section 4). Hence, there is a risk that the variation due to individual differences is larger than due to the treatment. Although this risk is considered a threat to the conclusion validity, it also helps to improving the ability to generalize the results of our experiment. There is an risk to internal validity due to we explored changes in the program elements through static analysis of the evolution of the code. We are not dealing with other classes of program elements or dependencies that may occur in the implementation of the feature.

Regarding the construct validity, the use of conditional compilation as the mechanism for implementing features in the source code might increase the number of feature dependencies when compared to other variability mechanisms. However, we argue that conditional compilation is one of the most widely used mechanisms to implement features [15, 28]. At last, the configurable systems analyzed might not be representative, to minimize this risk we evaluated configurable systems that come from heterogeneous application domains, and have been evaluated in previous research [22, 31]. Moreover, the scenarios might not represent relevant scenarios of evolution. To reduce this risk we considered all releases available in the repository for each project.

8 CONCLUSIONS

In this work, our goal was to unveil (i) the frequency of additions, exclusions, and modifications of feature dependencies, and in case of modifications (ii) explore the changes in program elements participating of the implementation of feature dependencies at the source-code level. We defined a catalog that list changes in program elements of feature dependencies in order to support our study. We found out that about 87% of the dependencies between subsequent versions are preserved whilst inclusions and exclusions of dependencies do not reach 13%. We found that approximately in 14% of those dependencies there is at least one change. Summed up with additions and deletions, approximately 26% of the dependencies are different between subsequent versions. This means that developers may have to deal with 1/4 of the dependencies on each evolution of the configurable system. Finally, we unveil that functions are 3x more prone to change than variables when participating in feature dependencies implementation. In addition, changes in the parameter list and function additions comprise more than 57%. These types of changes, together with others listed in our proposed catalog, indicate a high probability of impact on dependent features when a function is changed in the context of the implementation of a feature dependency. In this context, we plan to perform a further investigation with the goal of analyzing the impact of the changes touching feature dependencies in dependent features.

ACKNOWLEDGEMENTS

This work was funded by CNPq MSc. Scholarship (grant number 133822/2017-6).

REFERENCES

[1] Iago Abal, Jean Melo, Stefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wkasowski. 2018. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *IEEE Transactions on Software Engineering and Methodology* 26, 3 (2018), 1–34.

[2] Sven Apel, Don Batory, Christian Kstner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer Publishing Company, Incorporated.

[3] Sven Apel and Christian Kästner. 2009. Virtual Separation of Concerns - A Second Chance for Preprocessors. *Journal of Object Technology* 8, 6 (2009), 59–78.

[4] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wkasowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *7th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*. 1–8.

[5] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. 2018. APIDiff: Detecting API Breaking Changes. In *25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Tool Track*. 507–511.

[6] Bruno Cafeo, Elder Cirilo, Alessandro Garcia, Francisco Dantas, and Jaejoon Lee. 2016. Feature Dependencies As Change Propagators. *Information & Software Technology* 69, C (2016), 37–49.

[7] Bruno Cafeo, Francisco Dantas, Elder Cirilo, and Alessandro Garcia. 2013. Towards indicators of instabilities in software product lines: An empirical evaluation of metrics. In *4th International Workshop on Emerging Trends in Software Metrics (WETSoM)*. 69–75.

[8] Bruno Cafeo, Francisco Dantas, Alessandro Gurgel, Everton Guimarães, Elder Cirilo, Alessandro Garcia, and Carlos Lucena. 2012. Analysing the Impact of Feature Dependency Implementation on Product Line Stability: An Exploratory Study. In *26th Brazilian Symposium on Software Engineering (SBES)*. 141–150.

[9] Bruno Cafeo, Claus Hunsen, Alessandro Garcia, Sven Apel, and Jaejoon Lee. 2016. Segregating Feature Interfaces to Support Software Product Line Maintenance. In *15th International Conference on Modularity (MODULARITY)*. 1–12.

[10] Danny Dig and Ralph Johnson. 2006. How Do APIs Evolve&Quest; A Story of Refactoring: Research Articles. *J. Softw. Maint. Evol.* 18, 2 (2006), 83–107.

[11] Michael D. Ernst, Greg J. Badros, and David Notkin. 2002. An Empirical Analysis of C Preprocessor Use. *Transactions on Software Engineering* 28, 12 (2002), 1146–1170.

[12] Brady Garvin and Myra Cohen. 2011. Feature Interaction Faults Revisited: An Exploratory Study. In *22nd International Symposium on Software Reliability Engineering (ISSRE)*. 90–99.

[13] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Lessenich, Martin Becker, and Sven Apel. 2016. Preprocessor-based Variability in Open-source and Industrial Software Systems: An Empirical Study. *Empirical Software Engineering* 21, 2 (2016), 449–482.

[14] Kyo Kang, Sholom Cohen, James Hess, Willian Novak, and Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report. Carnegie-Mellon University Software Engineering Institute.

[15] Christian Kästner, Sven Apel, and Klaus Ostermann. 2011. The Road to Feature Modularity?. In *15th International Software Product Line Conference (SPLC)*. 1–8.

[16] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 805–824.

[17] Christian Kröher, Lea Gerling, and Klaus Schmid. 2018. Identifying the Intensity of Variability Changes in Software Product Line Evolution. In *22Nd International Systems and Software Product Line Conference (SPLC)*. ACM.

[18] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An Analysis of the Variability in Forty Preprocessor-based Software Product Lines. In *32Nd International Conference on Software Engineering (ICSE)*. 105–114.

[19] Jörg Liebig, Christian Kästner, and Sven Apel. 2011. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *10th International Conference on Aspect-oriented Software Development*. 191–202.

[20] Max Lillack, Christian Kästner, and Eric Bodden. 2014. Tracking Load-time Configuration Options. In *International Conference on Automated Software Engineering*. 445–456.

[21] Romero Malaquias, Márcio Ribeiro, Rodrigo Bonifácio, Eduardo Monteiro, Flávio Medeiros, Alessandro Garcia, and Rohit Gheyi. 2017. The Discipline of Preprocessor-based Annotations Does #Ifdef TAG N'T #Endif Matter. In *25th International Conference on Program Comprehension (ICPC)*. 297–307.

[22] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *38th International Conference on Software Engineering*. 643–654.

[23] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. 2015. The Love/Hate Relationship with the C Preprocessor: An Interview Study. In *29th European Conference on Object-Oriented Programming (ECOOP)*. 495–518.

[24] Flávio Medeiros, Márcio Ribeiro, and Rohit Gheyi. 2013. Investigating Preprocessor-based Syntax Errors. In *12th International Conference on Generative Programming: Concepts & Experiences (GPCE)*. 75–84.

[25] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. 2017. Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell. *Transactions on Software Engineering* (2017), 1–1.

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

[27] Rodrigo Queiroz, Leonardo Passos, Marco Tulio Valente, Claus Hunsen, Sven Apel, and Krzysztof Czarnecki. 2017. The Shape of Feature Code: An Analysis of Twenty C-preprocessor-based Systems. *Software & Systems Modeling* 16, 1 (2017), 77–96.

[28] Márcio Ribeiro, Paulo Borba, and Christian Kästner. 2014. Feature Maintenance with Emergent Interfaces. In *36th International Conference on Software Engineering (ICSE)*. 989–1000.

[29] Márcio Ribeiro, Felipe Queiroz, Paulo Borba, Társis Tolêdo, Claus Brabrand, and Sérgio Soares. 2011. On the Impact of Feature Dependencies when Maintaining Preprocessor-based Software Product Lines. In *10th International Conference on Generative Programming and Component Engineering (GPCE)*. 23–32.

[30] Márcio Ribeiro, Társis Tolêdo, Johnni Winther, Claus Brabrand, and Paulo Borba. 2012. Emergo: A Tool for Improving Maintainability of Preprocessor-based Product Lines. In *11th Annual International Conference on Aspect-oriented Software Development Companion (AOSD)*. 23–26.

[31] Iran Rodrigues, Márcio Ribeiro, Flávio Medeiros, Paulo Borba, Baldoino Fonseca, and Rohit Gheyi. 2016. Assessing fine-grained feature dependencies. *Information & Software Technology* 78 (2016), 27–52.

[32] Reimar Schröter, Norbert Siegmund, Thomas Thüm, and Gunter Saake. 2014. Feature-context Interfaces: Tailored Programming Interfaces for Software Product Lines. In *18th International Software Product Line Conference (SPLC)*. 102–111.

[33] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. 2012. *Experimentation in Software Engineering.* Springer Publishing Company, Incorporated.

[34] Larte Xavier, Aline Brito, André Hora, and Marco Valente. 2017. Historical and impact analysis of API breaking changes: A large-scale study. In *24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 138–147.