Feature Dependencies as Change Propagators: An Exploratory Study on Perfective Maintenance of Software Product Lines

Bruno B. P. Cafeo[a,d,*], Elder Cirilo[b], Alessandro Garcia[a], Francisco Dantas[c], Jaejoon Lee[d]

[a] *OPUS Research Group, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil*
[b] *Computer Science Department, Federal University of São João del-Rei, São João del-Rei, Brazil*
[c] *Computing Department, State University of Rio Grande do Norte, Natal, Brazil*
[d] *School of Computing and Communications, Lancaster University, Lancaster, UK*

Abstract

Context: A Software Product Line (SPL) is a set of software systems (products) that share common functionalities, so-called features. When different features are related, we consider this as feature dependencies. As the SPL evolves, dealing with feature dependencies in the source code in a cost- and effort-effective way is challenging. During the code maintenance of SPLs, features must eventually be updated when changes in other features affect them. In other words, any change in the code associated with a feature might imply changes to one or more dependent features. Thus, a main concern in product-line maintenance is to preserve consistency between related features by propagating changes (i.e., change propagation).
Objective: The objective of the study presented in this paper therefore is to examine the relation between feature dependency and change propagation. It is important to find whether there is a relation between feature dependency and change propagation. If so, it is also important to understand this relation in order to minimise the maintenance effort.
Method: We investigate change propagations through feature dependencies in perfective maintenance on five evolving medium-sized SPLs which encompass twenty-one representations of the SPL's evolution scenarios.
Results: The results have empirically confirmed for the first the close relation between feature dependency and change propagation. We also identified code parts that are more likely to cause change propagation when changed. The results also revealed that the extent of change propagation in SPL features might be higher than the one found in studies in modules of stand-alone programs (i.e., non-SPL). Finally, we also found a concentration of change propagation in a few feature dependencies.
Conclusion: In general, the results show that there is a relation between feature dependencies and change propagation, however such relation is not alike for all dependencies. This counter-intuitive conclusion indicates that (i) a general feature dependency minimisation might not ameliorate the change propagation, and (ii) characterising feature dependency properties must be analysed beforehand to drive maintenance effort to important dependencies.

Keywords: Software Product Line, Maintenance, Feature Dependency, Change Propagation

1. Introduction

Software product lines (SPLs) emerged as a prominent technology that aims to generate tailored programs (products) from a set of reusable assets, speeding up the software development process [1]. The goals of SPL-based development include managing variability, supporting automated product generation and facilitating reuse [2]. To achieve those goals, SPL-based development focuses on the software decomposition into modular units of functionality defined as *features*. Features are used to describe commonalities and variabilities of the products [3]. For example, in a mobile operating system, individual configurations share a common set of features (e.g. phone call and text message) but differ in other features (e.g. screen resolution or media management).

SPLs are usually developed incrementally in a way that new features are introduced on software products. A continual change effort is needed to keep the software up-to-date [4, 5]. In this context, SPL source code should be easy to evolve. Changes made during the evolution should affect the minimum of existing features as possible. In other words, the more features are affected by changes, the harder it becomes to evolve a SPL.

Considering the growing complexity and incremental development of SPLs, features naturally need to relate to other features to fulfil specific tasks [6]. These relationships are the so-called feature dependencies. In the source code, which is the focus of this paper, a feature dependency occurs whenever one or more program elements (e.g. methods or attributes) within the boundaries of a feature depend on elements external to that feature. A simple example is an attribute defined in one feature and used in another feature. The problem is that, in the presence of feature dependencies, any change in the SPL source code (e.g. addition of a new feature) might imply changes to one or more features. This implication of change is the result of the so-called change propagation [7].

*Corresponding author
Email address: bcafeo@inf.puc-rio.br (Bruno B. P. Cafeo)

Minimising change propagation stands out as one of the most desirable attributes of high-quality software. Change propagation involving many modules may cause an increase in the maintenance effort, and also may introduce errors in the system if dependencies are not well understood [8]. In addition, the SPL research community recently began discussing the impact of feature dependency as one of the indicators of SPL quality [9, 10, 11, 12]. However, to the best of our knowledge, there is a lack of empirical studies trying to relate these two concepts – i.e., feature dependencies and change propagation. It is neither obvious nor well understood to what extent and how they affect each other. This lack of knowledge may become a barrier for the adoption of SPLs.

In this paper, we aim to analyse the change propagation through feature dependencies during the perfective evolution [13] of a SPL implemented with conditional compilation (see Section 2). In particular, we want to understand basically (i) whether feature dependencies are related to change propagation, (ii) the extent of change propagation through paths of feature dependencies, and (iii) whether certain feature dependencies are involved more often in change propagation than other feature dependencies.To achieve our goals, we conducted an exploratory study on the evolution of one academic and four industrial medium-sized SPLs implemented with conditional compilation. The analysed SPLs comprise a total number of twenty-six releases[1] in sum (i.e., a total number of twenty-one evolution scenarios. Specifically, we analyse both feature dependencies and simultaneous changes (as an indicator of change propagation) in features during an evolution to identify the change propagation through paths of dependencies.

In summary, we make the following contributions:

- There is a strong relation between feature dependency and change propagation. In our study, this relation usually happens when there is a change in fragments of the feature code that are responsible for realising feature dependencies. So, a change in these fragments is more likely to demand a change in dependent features.
- Our analysis also evidenced a linear decay in the probability of change propagation in the path of feature dependencies. This means that the extent of change propagation in SPL features might be more severe than the extent in files of non-SPL systems (see [8]). Moreover, our data revealed that it is common to find transitive feature dependencies – i.e, a chain of dependencies – (77% of the SPL releases). Thus, several sampling-based analysis techniques for SPL that disregard larger feature combinations might be tuned in order to consider more features in these combinations.
- Our findings revealed an inequality in the distribution of change propagation along feature dependencies. This counterintuitive result basically indicates that a general minimisation of feature dependencies might not decrease the change propagation through paths of dependencies.

[1] Public distribution of a new upgraded version with improvements in functionality and bug fixing.

The remainder of this paper is structured as follows. Section 2 presents background on the main topics of this paper. A complete description of our study is provided in Section 3. Section 4 presents the data collected during the exploratory study. Section 5 presents an analysis of the data collected. Section 6 discusses the limitation of this work. Finally, Section 7 presents related 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 SPLs, features, feature dependency, change propagation and simultaneous change.

2.1. Software Product Lines and Features

A Software Product Line (SPL) is "a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission" [14]. SPLs enable the systematic construction of individual systems (i.e. products) with mass customisation. So, customers choose their products by selecting particular combinations of SPL's features. The use of SPLs bring significant improvements such as reduction of development costs, enhancement of quality and reduction of time-to-market [3, 14].

To gain all of these improvements, SPLs are organised and structured in terms of features. Features are the semantic units by which the commonalities and variabilities of the products are described [15]. There are common features that may be used among different products within a SPL. One can also have features by which different programs can be differentiated and defined. In this way, a SPL development process usually makes features explicit in requirements, design, code, testing and maintenance; i.e., across the entire life cycle [16].

Figure 1 illustrates a code snippet of how a feature is represented in the source code. In this example (Figure 1), we show part of the feature code WEIGHTED of a SPL to tailor graph data structures, including weighted edges and traversal strategies and algorithms. The selection of the feature WEIGHTED in a product allows to associate weight to graph edges. In this excerpt of code, we show the declaration of the attribute responsible for representing the weight of an edge in a graph (lines 3–5), and the methods responsible for getting and setting this weight (lines 7–19). It is worth to notice that the method `setWeight(int x)` (lines 11–18) only assigns positive values to weights.

The illustrated feature is implemented using the mechanism of conditional compilation. Conditional compilation is the most widely used mechanism to implement SPL features [17]. The preprocessor identifies the code that should be compiled or not based on preprocessor directives (i.e. `#ifdef` directives). Therefore, a feature is a set of program elements surrounded by preprocessor directives. In other words, directive tags encompass code associated with features. Then, to add the feature WEIGHTED in the final product, we set the corresponding directive tag to true for this feature. To remove WEIGHTED

```
01. public class Edge{
02.   ...
03.   #ifdef WEIGHTED
04.    int weight;
05.   #endif
06.   ...
07.   #ifdef WEIGHTED
08.    public int getWeight(){
09.     return weight;
10.    }
11.    public void setWeight(int w){
12.     if (w==0)
13.      weight = Integer.MAX_VALUE;
14.     else if (w<0)
15.      weight = -1 * w;
16.     else
17.      weight = w;
18.    }
19.   #endif
20.   ...
21. }
```

Figure 1: Part of the feature code WEIGHTED of a SPL of graph libraries.

from the final product, we set the corresponding directive tag to false. Moreover, it is important to notice here that features might be scattered through several modules of the source code and tangled with other feature code. In this way, the behaviour of change propagation (Section 2.3) in feature code may be totally different from change propagation in programming modules (e.g. classes).

2.2. Feature Dependency

Features depend on each other to fulfil specific SPL requirements [6]. In the source code, a feature dependency occurs whenever one or more program elements 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 [18]. In other words, feature dependencies are established by means of structural dependencies in the source code between elements of different features. It is important to highlight that structural dependency encompasses different ways of realising a feature dependency, such as control-flow dependencies or inheritance. In this paper, we consider the cases of realisation of feature dependencies in the source code as described in Section 3.2.

Figure 2 illustrates an example of the realisation of a feature dependency in the SPL for generating tailored graph data structures. The code snippet in Figure 2 illustrates part of the feature code DIJKSTRA. This feature solves the shortest path problem for a non-negative edge path of a weighted directed graph [19]. The feature dependency of this example is realised by the method call `getWeight()` (highlighted in line 05). The method `getWeight()` is a method that belongs to feature WEIGHTED (lines 8–10, Figure 1). So, a part of the DIJKSTRA feature code depends on elements of other feature, thus establishing a dependency between the features WEIGHTED and DIJKSTRA.

```
01. public class Algorithms{
02.   ...
03.   #ifdef DIJKSTRA
04.    ...
05.    calculatePath(obj.getWeight());
06.    ...
07.   #endif
08.   ...
09. }
```

Figure 2: Part of the feature code DIJKSTRA with the realisation of a feature dependency.

In this study, we also consider transitive dependencies. A transitive feature dependency happens whenever a feature C depends on feature B, and B is in turn dependent on a feature A, then C depends on A by transitivity (a.k.a. indirect dependency). For instance, let us say a feature AUXILIARY, responsible for providing additional support to calculate the shortest path in a graph, depends on feature DIJKSTRA. In this case, the feature AUXILIARY also depends on feature WEIGHTED due to the dependency between DIJKSTRA and WEIGHTED. So, from hereafter, feature dependency refers to either a direct dependency or a transitive dependency. Moreover, there is a distance between AUXILIARY and WEIGHTED in the path of feature dependencies linking these features. So, since we consider that a feature dependency is a directional relationship, we say the distance between AUXILIARY and WEIGHTED is two, and the distance between WEIGHTED and AUXILIARY is zero.

2.3. Simultaneous Change and Change Propagation

Let us suppose an evolution where the feature BELLMAN-FORD is included (lines 9–13, Figure 3). This feature, similarly to feature DIJKSTRA, implements an algorithm that computes the shortest path in a weighted directed graph. However, this feature is capable of handling graphs with negative weight edges [20].

Figure 3 illustrates the inclusion of part of the BELLMAN-FORD feature code. It is important to notice that the feature responsible for implementing the Bellman-Ford algorithm also depends on feature WEIGHTED, similarly to feature DIJKSTRA. The dependency between BELLMAN-FORD and WEIGHTED happens by means of the method `getWeight()`, highlighted in line 11.

Due to all capabilities of the Bellman-Ford algorithm, the developer needs to allow the assignment of negative weight values to edges in the SPL. So, it is necessary to change the method `setWeight(int x)` in order to assign negative weight values to edges. Figure 4 shows the method `setWeight(int x)` after the change.

Moreover, since the Dijkstra algorithm is not capable of handling negative weight values of edges, a change in the DIJKSTRA feature code also must be made to treat a possible negative value. In our example, the developer decided to consider the weights as unsigned values. Figure 5 illustrates a possible change in the code of the feature DIJKSTRA (lines 3–10).

In this example, it is important to highlight and understand the simultaneous changes that happened due to the presence of

```
01. public class Algorithms{
02.  ...
03.  #ifdef DIJKSTRA
04.   ...
05.   calculatePath(obj.getWeight());
06.   ...
07.  #endif
08.  ...
09.  #ifdef BELLMAN-FORD
10.   ...
11.   calculatePath(obj.getWeight());
12.   ...
13.  #endif
14.  ...
15. }
```

Figure 3: Part of the code of the added feature BELLMAN-FORD with the realisation of a feature dependency.

```
01. ...
02. #ifdef WEIGHTED
03.  ...
04.  public void setWeight(int w){
05.   if (w==0)
06.    weight = Integer.MAX_VALUE;
07.   else
08.    weight = w;
09.  }
10. #endif
11. ...
```

Figure 4: Change in the method `setWeight(int x)`.

a feature dependency between features WEIGHTED and DIJKSTRA. Simultaneous change is an event where different features have changed in the same evolution. The inclusion of the feature BELLMAN-FORD in a SPL evolution caused a change in the feature WEIGHTED to adjust the SPL to manage negative values of edge weights. Moreover, there was a change in feature DIJKSTRA. So, code of features WEIGHTED and DIJKSTRA were changed from one release to the next one.

The change in feature DIJKSTRA happened because there was a change propagation through the dependency during the evolution of the SPL. Change propagation are the changes required to other entities of a software system to ensure the consistency of the already existing assumptions after a change in a particular entity [21]. In other words, the addition of the feature BELLMAN-FORD demanded a change in feature WEIGHTED. As a consequence, the change in feature WEIGHTED demanded a change in the dependent feature DIJKSTRA to ensure the consistency of the SPL. So, we can say that features WEIGHTED and DIJKSTRA changed simultaneously in the evolution due to a change propagation from feature WEIGHTED to feature DIJKSTRA.

In general, when features exhibit dependencies, a change propagation may happen. Thus, since these features are changed together in the same evolution, one can say that a simultaneous change can indicate a change propagation when it involves dependent features.

```
01. public class Algorithms{
02.  ...
03.  #ifdef DIJKSTRA
04.   int w = obj.getWeight();
05.   ...
06.   if (w<0)
07.    w = -1*w;
08.   calculatePath(w);
09.   ...
10.  #endif
11.  ...
12.  #ifdef BELLMAN-FORD
13.   ...
14.   calculatePath(obj.getWeight());
15.   ...
16.  #endif
17.  ...
18. }
```

Figure 5: Change in the feature DIJKSTRA.

3. Research Setting

This section presents the research questions and motivates the relevance of this study, followed by a definition of the statistical analysis designed to answer the questions. Also the target systems are presented. The section is concluded with a description of the evaluation procedures. The design of our study was inspired in the work of Geipel and Schweitzer [8] because such design allows to better understand how and to what extent feature dependency and change propagation affect each other. However, they analysed only simultaneous changes involving classes rather than features of SPL. The method used in their work was adapted for the context of features and SPLs in this study with several changes in formulas and experimental design. The reuse and adaptation of their study design would enable us to contrast their results with ours (Section 5).

3.1. Research Questions

During the evolution of a SPL, it is important to minimise changes in already existing features. Changes in several features may be costly to the evolution of the SPL. However, it is not uncommon to have changes in some existing features during an evolution. Moreover, several of these changes happen in dependent features. Thus, it is important to find whether there is a relation between feature dependency and change propagation. If so, it is also important to understand this relation in order to minimise the evolution effort.

In this context, one can hypothesise: if there are simultaneous changes in dependent features, they may be caused by a change propagation along the path of dependencies between features. In other words, a simultaneous change in the presence of a feature dependency may indicate a change propagation through this dependency. So, it should be a relation between feature dependency and change propagation. Thus, the first research question addresses whether this is the case: *[RQ1] Are dependent features more likely to change simultaneously?*

If there is a relation between feature dependency and change propagation, we can assume that changes might be propagated along the path of feature dependencies. In other words, a

change in one feature may cause a change cascade along the path of feature dependencies even to distant features. In this context, the more features are affected by changes, the harder it may be to evolve a SPL. So, it is important to understand the extent of the change propagation through the dependency network, since programmers often inspect the code of direct neighbour features at best. Thus, the second research question is concerned about the relation between change propagation and distance between features in the path of dependencies: *[RQ2] What is the relation between probability of simultaneous change and the distance between features in the path of dependencies?*

The last research question was inspired by the assumption that all feature dependencies are assumed to equally impact change propagation [9, 18, 22]. In other words, these studies assume that each dependency between features incurs the same impact on change propagation, whatever be their characteristics.

The equality regarding the impact of feature dependencies on change propagation has important ramifications. If the majority of feature dependencies equally impact on change propagation, they should be reduced. On the other hand, if only specific dependencies matter, ad hoc reduction of dependencies might not reduce change propagation. In this case, an identification of feature dependencies that causes more change propagation would ameliorate this problem. Thus, the third research question of this paper is concerned about the equality of the impact of feature dependencies on change propagation: *[RQ3] Are feature dependencies equally involved in the change propagations?*

3.2. Feature Dependency

Two mathematically equivalent notations are commonly used to represent dependencies between elements: the graph notation and the adjacency matrix notation. In this paper, we chose the adjacency matrix notation to represent feature dependencies because it is the most widely used notation to represent software dependencies [8].

We refer to the feature dependency matrix as FD, where each feature appears in one row and one column of the matrix. $FD_{a,b} \geq 1$ means that a (row) depends on b (column). Therefore, $FD_{a,b} = 0$ is interpreted as independence. There is a dependency between features a and b if: (i) a references an attribute of b, or (ii) a calls a method of b. We set $FD_{a,b} = 1$ if at least one of these cases happen. To add more representative power to this approach, we also represent the distance d of transitive dependencies between features in the matrix entries. For instance, let us suppose that a feature A depends on feature B, and feature B depends on feature C. In this case, $FD_{a,b} = 1$, $FD_{b,c} = 1$ and $FD_{a,c} = 2$. Therefore, the value of FD represents the distance d between two features. The values of distance between features in paths of feature dependencies are calculated by using the Floyd-Warshal algorithm [23]. Moreover, it is worth to notice that FD is an asymmetric matrix since we consider dependency as a directional relationship.

To illustrate the extraction of feature dependencies from the code, we use the same example shown in Section 2. Considering the situation previously described for features WEIGHTED, DIJKSTRA, BELLMAN-FORD and AUXILIARY, we have the FD matrix presented in Figure 6. For instance, the dependency of DIJKSTRA on WEIGHTED is represented by $FD_{D,W} = 1$.

	WEIGHTED (W)	DIJKSTRA (D)	BELLMAN-FORD (BM)	AUXILIARY (A)
WEIGHTED (W)	0	0	0	0
DIJKSTRA (D)	1	0	0	0
BELLMAN-FORD (BM)	1	0	0	0
AUXILIARY (A)	2	1	0	0

Figure 6: Matrix representing feature dependencies of a SPL.

To perform this extraction automatically, two authors of this paper used the tool CIDE [24, 25] that supports mapping of features in the source code. CIDE was also used to relate program elements to features. The CIDE output was used by an extension of the tool GenArch+ [26] to generate the matrix of feature dependencies. GenArch+ uses an AST (Abstract Syntactic Tree), based on the source code, with information about features in its nodes to extract feature dependencies. Each program element (or block of program elements) is represented by a node in the tree, and it is associated to one or more features. Thus, when one program element from one feature is used by a program element of another feature directly, we consider this as a feature dependency. The transitive dependencies of the feature dependency matrix were calculated by a program implemented for the purpose of this study. This program implement the Floyd-Warshal algorithm [23] in order to find the distance between features in a path of feature dependencies. All tools are available in the website of this study [27].

It should be noted that each SPL release has a FD matrix. However, since we are interested in analysing the evolutions, we focus on the latest snapshot of FD (last release) which aggregates information of all dependencies.

3.3. Simultaneous Changes

Similarly to feature dependencies (Section 3.2), we use a matrix structure notation to represent the simultaneous changes of features in the evolutions of the SPL. Moreover, we model our simultaneous changes matrix adapting the simultaneous change model presented in the work of Geipel and Schweitzer [8]. This matrix represents the dynamic view of the SPL evolution based on simultaneous changes. Its entries indicate the number of times the features have been changed simultaneously. Let us refer to this matrix as C, and the event of features being *"changed at the same time"* in an evolution as a simultaneous change.

To construct C, we need (i) the set of features, and (ii) the evolution scenario which record changes of the features. Let us use n to denote the number of the feature and m to refer to the number of evolutions. An event in the SPL evolution can be expressed as a n-dimensional vector $\vec{e}$. Each entry shows in binary form whether a feature has been modified. For instance, in the first evolution of a SPL we have three features. The evolution scenario $\vec{e}_1 = (101)^T$ indicates that features one and three were modified simultaneously in the first evolution. Thus, each $\vec{e}$ corresponds to one SPL evolution.

The evolution matrix of a SPL can be written by considering each $\vec{e}$ as a column of the evolution matrix E of size $n \times m$:

$$E = (\vec{e}_1 \vec{e}_2 \ldots \vec{e}_m) \quad (1)$$

To have a matrix representing the whole evolution scenario of a SPL, we need to multiply E with its transposed E^T, and thus the simultaneous changes matrix C is derived:

$$C = EE^T \quad (2)$$

The matrix C has dimension $n \times n$ and indicates how many times each feature has been modified simultaneously with other feature. An entry $C_{a,b} = 4$, for instance, indicates that features a and b have been changed four times together considering all SPL evolutions. Note that C, in contrast to FD, is a symmetric matrix.

We consider only existing features changed during an evolution in our n-dimensional vector $\vec{e}$. For instance, in our example of SPL evolution presented in Section 2, we do not consider the addition of the feature BELLMAN-FORD as a change in this feature for this specific evolution. Thus, we have a four-dimensional vector $\vec{e}_1 = (1100)^T$ to represent the features changed in the evolution where BELLMAN-FORD was added. The values of the vector corresponds to the following order of features: WEIGHTED, DIJKSTRA, BELLMAN-FORD and AUXILIARY. So, the first two values indicate that features WEIGHTED and DIJKSTRA were changed in the same evolution. Moreover, the last two vector positions indicate that the features BELLMAN-FORD and AUXILIARY were not changed in this specific evolution. If we consider only this change and this evolution, C would be like the matrix presented in Figure 7.

	WEIGHTED (W)	DIJKSTRA (D)	BELLMAN-FORD (BM)	AUXILIARY (A)
WEIGHTED (W)	0	0	0	0
DIJKSTRA (D)	1	0	0	0
BELLMAN-FORD (BM)	0	0	0	0
AUXILIARY (A)	0	0	0	0

Figure 7: Matrix representing simultaneous changes in all evolutions of a SPL.

To perform this extraction automatically, two authors of this paper used an extension of the tool GenArch+ [26] to analyse the evolutions and extract the change in the features. The simultaneous changes matrix was calculated by a program implemented for the purpose of this study. It should not be left unmentioned that the tools used in this experiment are available in the website of the study [27].

3.4. The Target Systems

As target systems, we selected five product lines available in open repositories and managed by different developers. All of them have been developed in Java with conditional compilation for different purposes and, besides the Graph Product Line (see below), all of them are non-academic SPLs. In the following we describe each target system:

- **Berkeley DB.** It is an open source database engine that can be embedded as a library into applications [28, 29]. The core features represent basic data management and transactional behaviour support whilst the variabilities include logging and statistics, among others. The evolution scenarios comprise the addition of optional features such as logging and file handle cache.

- **Mobile RSS.** It is a portable RSS reader for mobile phones on the Java ME platform [29, 30]. Mobile RSS basically provides features to parse, browse and read RSS feeds. Moreover, for instance, there are features to deal with compatibility issues of mobile phones and features for logging. The evolution scenarios comprise the addition of optional features.

- **Lampiro.** It is a messaging client for mobile devices based on a protocol called XMPP [29, 31, 32]. It is possible to connect with contacts of ICQ, Gtalk and Windows Messenger. There are many features including compression, encryption and languages supported. The evolution scenarios comprise the addition of different features.

- **Graph Product Line (GPL).** It is a SPL of graph libraries that allows programmers to tailor graph data structures, including weighted and directed edges as well different traversal strategies and algorithms [33]. The evolution scenarios comprise the addition of new features.

- **Java Chat.** It is a simple chat application with features such as GUI, encryption and logging [29]. The evolution scenarios comprise the inclusion of optional features.

Table 1 shows general data about the target SPLs, such as lines of code (KLOC), number of features (# of Features), number of preprocessor directives implementing features found in the source code (# of IFDEFS) and number of SPL releases (# of releases).

Table 1: General information about the target SPLs

	KLOC	# of Features	# of IFDEFS	# of Release
Berkeley DB	39	56	4051	6
Mobile RSS	18	31	2990	6
Lampiro	31	18	164	6
GPL	1	26	582	4
Java Chat	0.6	9	105	4

We focused only in evolutions with perfective maintenance [13]. A perfective maintenance comprises enhancements intended to make the system better – i.e. mainly the addition of new features in the context of SPLs [22, 34, 35]. Evolutions with perfective maintenance alter the intended outward semantics of the system, thus justifying the creation of a new SPL release. Moreover, more than 60% of the maintenance tasks are associated to a perfective maintenance [36]. In other words, it is the most common maintenance type in software evolution.

Based on the respective original systems from open repositories [24, 28, 29, 30, 31], three authors created representations of evolutions scenarios comprising only addition of

new features. The representation of the evolution scenarios were created based on information of the open repositories [24, 28, 29, 30, 31], source code analysis and information about evolution scenarios extracted from studies about SPL evolution [22, 34]. In this way, each evolution of our target SPLs comprises one or more type of change, such as inclusion of optional features, inclusion of alternative features, inclusion of mandatory features, and implementation of new SPL constraints due to the inclusion of new features. For instance, the evolution of BerkeleyDB involves the inclusion of optional features such as logging and file handle cache, whereas GPL evolution comprises the addition of mandatory and alternative features

These evolutions scenarios were retrospectively implemented by undergraduate and postgraduate students based on the latest release of each SPL. In other words, based on the representation of the evolutions created by some authors of the papers, the students could isolate and simulate in the source code evolutions considering only the addition of new features. Good design principles and practices were used, enforced, and reviewed throughout the creation of all the SPL releases. These practices were applied to ensure that the purpose of each feature was achieved as expected. In all the cases, the evolution scenarios were also reviewed by experts in the field. The source code of all target SPL are available in the website of the study [27].

3.5. The Evaluation Procedures

This section describes our evaluation procedures to answer the research questions presented in Section 3.1.

3.5.1. The Relation Between Feature Dependency and Change Propagation (RQ1)

To understand the relation between feature dependency and change propagation, we make a twofold comparison: (i) probability of simultaneous changes in dependent features (point of view of simultaneous changes), and (ii) probability of dependent features with simultaneous changes (point of view of dependencies). We are using probabilistic analysis because we want to analyse typical behaviour of feature dependencies on change propagation. Specifically, we are using conditional probability to analyse the data and answer RQ1. A conditional probability measures the probability of occurrence of an event given that (by assumption, presumption, assertion or evidence) another event has occurred [37].

Simultaneous changes view. We express the fact that two features (a and b) have been changed at least once simultaneously by $C_{a,b} \geq 1$. Moreover, the fact that there is a dependency between two features (a and b) is expressed by $FD_{a,b} \geq 1$. So, in our first comparison, to measure the impact of feature dependencies on simultaneous change we use the conditional probability $P_{FD} = P(C_{a,b} \geq 1 | FD_{a,b} \geq 1)$ given $a \neq b$. In other words, the probability that two different features have been changed at least once simultaneously, given that there is a dependency between them. P_{FD} is calculated as follows:

$$P_{FD} = \frac{P(FD_{a,b} \geq 1 \wedge C_{a,b} \geq 1)}{P(FD_{a,b} \geq 1)} \quad (3)$$

This means that we divide the number of simultaneous changes happened in pairs of features involved in a feature dependency by the total number of feature dependencies.

As a reference, we compute the conditional probability $P_{\neg FD} = P(C_{a,b} \geq 1 | FD_{a,b} = 0)$ given $a \neq b$. This means that we are interested in calculating the probability that two different features have been modified together at least once given that they are independent. $P_{\neg FD}$ is calculated as follows:

$$P_{\neg FD} = \frac{P(FD_{a,b} = 0 \wedge C_{a,b} \geq 1)}{P(FD_{a,b} = 0)} \quad (4)$$

This means that we divide the number of simultaneous changes happened in pairs of features that are not involved in a feature dependency by the total number of pairs of features that do not present a dependency.

The comparison of P_{FD} with $P_{\neg FD}$ reveals the possible degree of correlation between feature dependency and simultaneous change in terms of dependency.

Feature dependency view. In our second comparison, we are interested in analysing the other way: the probability of existing a feature dependency given that there is a simultaneous change or not. As aforementioned, we express the fact that there is a dependency between two features (a and b) by $FD_{a,b} \geq 1$ and that two features (a and b) have been changed at least once simultaneously by $C_{a,b} \geq 1$. A measure for the relation between simultaneous change and feature dependency is the conditional probability $P_C = P(FD_{a,b} \geq 1 | C_{a,b} \geq 1)$ given $a \neq b$ meaning the probability that a feature dependency exists given that two different features have been changed at least once simultaneously. P_C is calculated as follows:

$$P_C = \frac{P(C_{a,b} \geq 1 \wedge FD_{a,b} \geq 1)}{P(C_{a,b} \geq 1)} \quad (5)$$

This means that we divide the number of simultaneous changes happened in pairs of features involved in a feature dependency by the total number of pairs of features which changed simultaneously.

As a reference, we compute the conditional probability $P_{\neg C} = P(FD_{a,b} \geq 1 | C_{a,b} = 0)$ given $a \neq b$. This means that we are interested in calculating the probability that a feature dependency exists in the absence of a simultaneous change. $P_{\neg C}$ is calculated as follows:

$$P_{\neg C} = \frac{P(C_{a,b} = 0 \wedge FD_{a,b} \geq 1)}{P(C_{a,b} = 0)} \quad (6)$$

This means that we divide the number of simultaneous changes happened in pairs of features involved in a feature dependency by the total number of pairs of features which have not changed simultaneously during the SPL evolution.

The comparison of P_C with $P_{\neg C}$ reveals the possible degree of correlation between feature dependency and simultaneous change from the point of view of dependencies.

3.5.2. Feature Dependency Distance (RQ2)

Features might propagate changes along the path of feature dependencies. In this context, one can assume that there is

a decreasing probability of simultaneous changes of features as the distance between features increases in the path of feature dependencies. In this context, we extend the concept of P_{FD} to $P_{FD}(d)$, giving the probability that two features connected by a dependency with distance d are changed together at least once. So, if changes propagate along a path of dependencies, which function P_{FD} in d would indicate the probability of change propagation?

It is known that, when only one dependent variable is being modelled, a scatterplot can suggest the form and strength of the relationship between variables. In our study, the scatterplots of all target systems suggest that there is a relationship between d and P_{FD}, and this relationship can be approximated as a linear function. Consequently, the most simple approximation for $P_{FD}(d)$ is:

$$P_{FD}(d) = -ad + b \tag{7}$$

We add a constant a with a minus sign to the equation because (i) we are supposing that there is a decreasing probability of change propagation related to distance, and (ii) as a rule, the constant term is always included in the set of regressors, which in our case is d. Moreover, since $P_{\neg FD}$ may be nonzero, we also add an intercept b to the equation.

Having the possible equation that delineates $P_{FD}(d)$, we need to test the goodness of fit of the statistical model proposed regarding our data. Measures of goodness of fit typically summarise the discrepancy between observed values and the values expected under the model in question. Since our statistical model proposed is a linear function, we fit the observed data $P_{FD}(d)$ to equation 7 with the ordinary least squares method. Ordinary least squares is a method for estimating the unknown parameters in a linear regression model. This method minimises the sum of squared vertical distances between the observed responses in the dataset and the responses predicted by the linear approximation. To check whether the model explains the patterns found in the data, the quality of the fit needs to be quantified. To measure the quality of fit, we consider the adjusted coefficient of determination ($adj.R^2$).

The coefficient of determination of a linear regression model is the quotient of the variances of the fitted values and observed values of the dependent variable. If we denote y_i as the observed values of the dependent variable, $\bar{y}$ as its mean, and $\hat{y}_i$ as the fitted value, then the coefficient of determination is:

$$R^2 = \frac{\sum (\hat{y}_i - \bar{y})^2}{\sum (y_i - \bar{y})^2} \tag{8}$$

The adjusted coefficient of determination ($adj.R^2$) of a linear regression model is defined in terms of the coefficient of determination (equation 8). R^2 is adjusted to account for the residual degrees of freedom (number of observations minus the number of fitted coefficients). If we denote n as the number of observations in the dataset, and p as the number of independent variables, the adjusted coefficient of determination is:

$$adj.R^2 = 1 - (1 - R^2)\frac{n-1}{n-p-1} \tag{9}$$

The adjusted coefficient of determination ($adj.R^2$) takes a value between -1 and 1, with values of $adj.R^2$ close to 1 denoting that the model fits the data well. In other words, high values of $adj.R^2$ provide evidence for the existence of change propagation, and thus a relation between probability of simultaneous change and distance along the paths of dependencies.

We are using the adjusted coefficient of determination to quantify the quality of the fit because (i) it is bound between -1 and 1 making the comparison between the target systems easier than unbound measures, and (ii) it overcomes specific problems of the coefficient of determination (R^2) in order to provide additional information by which we can evaluate our regression model's explanatory power.

3.5.3. Equality on the Impact of Change Propagation (RQ3)

Given the number of feature dependencies per release of SPL (e.g., 235 in the last release of MobileRSS), there are too many data points to permit a comparison of all feature dependencies individually. Thus, to address our research question, we apply data aggregation [38] using concentration statistics. The idea is to analyse the equality of the impact of feature dependencies on change propagation. This analysis allows us to make statements such as "10% of the feature dependencies are responsible for over 70% of change propagation".

As a statistic concentration, we adopt a method to analyse and visualise income inequalities in a population of a country called Lorenz inequality or Lorenz curve [38, 39]. In this paper, we use it to analyse the concentration of change propagation in feature dependencies. For a more in-depth description of Lorenz inequality, the reader may refer to the original work of Lorenz [39].

First, we need the number of simultaneous changes for each dependency. Let us refer to this set as Π:

$$\Pi = \{C_{a,b} : FD_{a,b} \geq 1 \wedge C_{a,b} \geq 1\} \tag{10}$$

The next step is to normalise Π, so the entries of the result π sum up to one:

$$\pi_n = \Pi_n / \sum_{l=1}^{|\Pi|} \Pi_l \tag{11}$$

After that, the entries of π must be rearranged in ascending order:

$$m < n \Rightarrow \pi_m < \pi_n \tag{12}$$

Finally, the Lorenz curve $L(x)$ with $0 \leq x \leq 1$ is calculated by cumulating the first x percent of the elements of π:

$$L(x) = \sum_{n=0}^{\lfloor x*|\pi| \rfloor} \pi_n \tag{13}$$

$L(x)$ is the percentage of simultaneous changes accumulated in the x percent least active feature dependencies. An equal distribution leads to $L(x) = x$. To compare the Lorenz concentration between the target SPLs we compress it to one number named Gini coefficient [40]. The Gini coefficient indicates

the degree of distributional inequality of simultaneous changes amongst the feature dependencies of a SPL. Let us refer to the Gini coefficient as g:

$$g = 1 - 2\int_0^1 L(x)\,\mathrm{d}x \tag{14}$$

The Gini coefficient takes a value between 0 and 1, with $g = 0$ denoting perfect equality meaning that x percent of the feature dependencies are responsible for x percent of the simultaneous changes. Conversely, $g = 1$ denotes perfect inequality with only one feature dependency for 100 percent of the cumulative feature dependencies. By using Gini coefficient we are able to compare different concentrations since such coefficient represents a concentration in a scalar value.

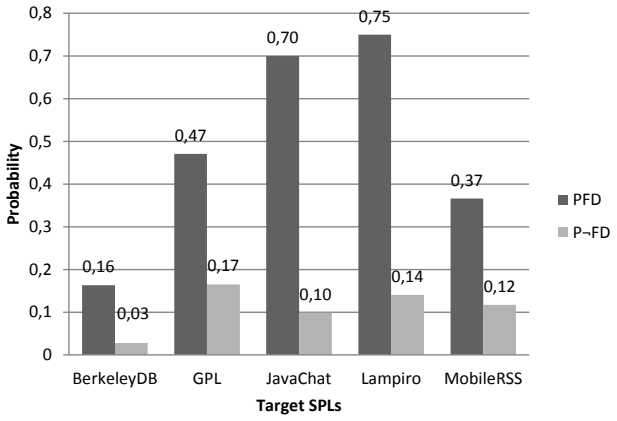

Figure 8: Comparison between P_{FD} and $P_{\neg FD}$.

4. Results

This section presents the results of the analysis described in Section 3. Section 4.1 describes the results of the probabilities extracted to answer research question RQ1. Section 4.2 checks whether the statistical model proposed to represent change propagation along the path of dependencies represent the patterns found in our data in order to answer RQ2. Finally, addressing RQ3, Section 4.3 shows the results obtained regarding the concentration of change propagation in certain feature dependencies.

4.1. Feature Dependency and Change Propagation

As pointed out in Section 3, the probabilities P_{FD}, $P_{\neg FD}$, P_C and $P_{\neg C}$ can provide evidence of the relation between feature dependency and change propagation. Table 2 shows the probability values for P_{FD} and $P_{\neg FD}$ for the five SPLs. These values indicate the probability that two features change simultaneously in the presence of feature dependency (P_{FD}) or in the absence of feature dependency ($P_{\neg FD}$). The comparison between these values indicates the influence of a feature dependency in the occurrence of a possible change propagation. It is important to notice that the amount of feature dependencies or simultaneous changes affect the probabilities results, but not the difference between them. A change in the amount of these variables should affect both the conditional probability and its reference probability. Since we are comparing a conditional probability with its reference, the difference between them is supposed to be the same (or similar) when the amount of one or both variables change.

Looking at the values of P_{FD} and $P_{\neg FD}$, we can notice that P_{FD} ranges between 0.16 and 0.75, and $P_{\neg FD}$ is less than 0.16 for all SPLs. This basically means that it is likely that change propagation happens when a feature dependency exists. This superiority of P_{FD} against $P_{\neg FD}$ can be seen in the mean of these values ($P_{FD} = 0.504$ vs. $P_{\neg FD} = 0.114$). Moreover, this behaviour can be observed in all SPLs analysed as shown in Figure 8.

In Figure 8 we can notice that the probability of simultaneous changes in dependent features (P_{FD}) is much higher than the probability in independent features ($P_{\neg FD}$). The smallest difference between P_{FD} and $P_{\neg FD}$ is observed in GPL. In this case, the chances of simultaneous changes in dependent features is almost three times higher than simultaneous changes in independent features ($P_{FD} = 0.47$ vs. $P_{\neg FD} = 0.17$). Additionally, the most significant difference is in JavaChat where the probability is seven times higher ($P_{FD} = 0.70$ vs. $P_{\neg FD} = 0.10$).

Table 2 also shows the values of P_C and $P_{\neg C}$. These values show the probability that a feature dependency exists given that there is a simultaneous change (P_C) or not ($P_{\neg C}$) in features during SPL evolution. By comparing these values it is possible to conclude whether the occurrence of simultaneous changes is an indicative of the existence of a dependency between features. P_C ranges between 0.062 and 0.722 while $P_{\neg C}$ ranges between 0.003 and 0.275. Except for the Lampiro SPL, all values of P_C are greater than the highest value of $P_{\neg C}$ (0.275 for BerkeleyDB). The lowest value of P_C is 0.378 if Lampiro is discarded. Lampiro present odd results due to the way some dependencies between features are established. Our approach captures structural dependencies in the source code as described in Section 3.2. However, there are some relationships between features in Lampiro SPL that are not being considered by our definition. In this way, we are counting simultaneous changes between some of these features, however our approach do not consider a dependency between them. Nevertheless, the mean values presented in Table 2 also show that there is a considerable difference between P_C and $P_{\neg C}$. This difference can be observed in each SPL as shown in Figure 9.

As we can see in Figure 9, there is a clear difference between P_C and $P_{\neg C}$ in terms of values in each SPL. The lowest difference is presented in GPL where P_C is almost three times higher than $P_{\neg C}$ (0.38 and 0.12, respectively), while the most significant difference with almost thirteen times is encountered in JavaChat ($P_C = 0.50$ vs. $P_{\neg C} = 0.040$), excluding the unusual results of Lampiro.

4.2. Distance and Change Propagation

In Section 3.5.2, we observed that there is a decreasing probability of change propagation as the distance between features increases in the path of dependencies. In other words,

Table 2: Summary of Statistical Analysis

SPL Projects	P_{FD}	$P_{\neg FD}$	P_C	$P_{\neg C}$	$Adj.R^2$	g
BerkeleyDB	0.163	0.028	0.722	0.275	0.908	0.862
Graph Product Line (GPL)	0.471	0.165	0.378	0.119	0.799	0.595
JavaChat	0.700	0.099	0.500	0.045	1.000	0.375
Lampiro	0.750	0.141	0.062	0.003	1.000	0.250
MobileRSS	0.366	0.117	0.524	0.202	0.958	0.657
mean	0.504	0.114	0.425	0.123	0.933	0.535

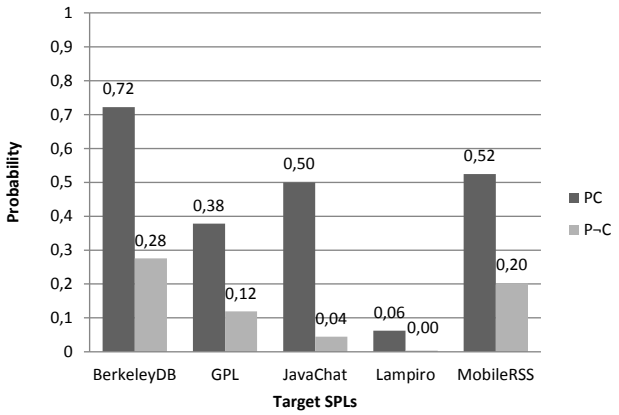

Figure 9: Comparison between P_C and $P_{\neg C}$.

if changes propagate along the dependency network, there is a relationship between d and P_{FD}, and we assume that this relationship can be approximated as a linear function. Figure 10 presents a scatterplot of the relationship between d and P_{FD} in BerkeleyDB (solid black dots) and a reference graph with linear trend (grey dashed line). The linear trend of the relationship between d and P_{FD} in BerkeleyDB is not an exception amongst the SPLs analysed – i.e., all SPLs presented the same linear trend between d and P_{FD}.

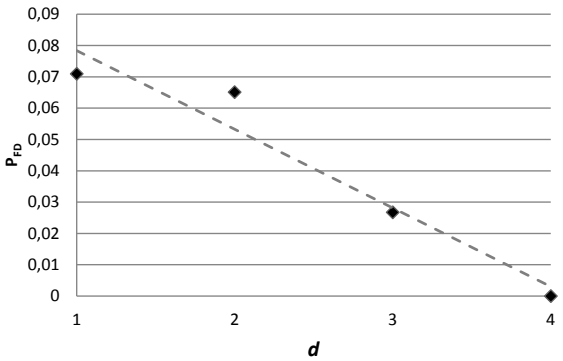

Figure 10: Relationship between d and P_{FD} in BerkeleyDB.

As we propose a statistical linear model to delineate $P_{FD}(d)$, we test the goodness of fit of the statistical model proposed regarding our data using the adjusted coefficient of determination ($adj.R^2$). As indicated by the high values of $adj.R^2$ in Table 2, the linear model proposed describes well $P_{FD}(d)$. BerkeleyDB has an $adj.R^2$ of 0.908. Moreover, all SPLs have an $adj.R^2$ larger than 0.7, evidencing the existence of change propagation, and thus a relation between probability of simultaneous change and distance along the paths of dependencies.

It is important to note that some SPLs have values for $adj.R^2$ equal to 1 because the maximum values of d for these SPLs are two. In other words, when $d = 2$ we obtain a linear relationship between d and P_{FD}.

4.3. Inequality in the Distribution of Change Propagation

Figure 11 shows in dashed grey the Lorenz curve for BerkeleyDB. As a reference, the solid black line marks the line of equality. It can be seen that the Lorenz curve is strongly bent and that less than 20% of the dependencies concentrate 100% of the change propagation.

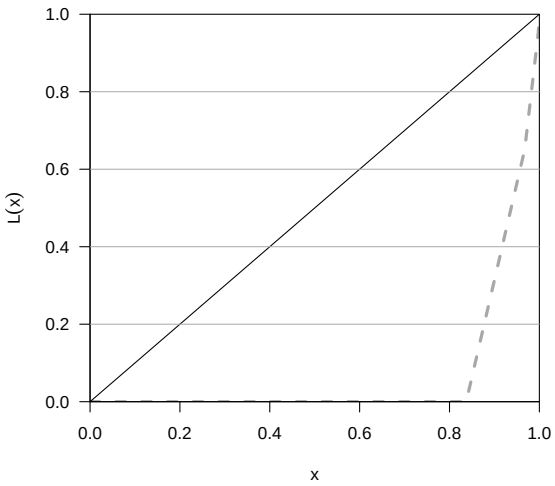

Figure 11: Concentration of change propagation through paths of feature dependencies in BerkeleyDB.

This behaviour of inequality in the distribution of simultaneous changes can be confirmed by the high value of the Gini Coefficient g for BerkeleyDB ($g = 0.862$). The situation is similar for most of the other projects. Table 2 lists the Gini Coefficients g for all SPLs, and throughout the sample the concentration is very high; on average of 0.535. It is worth to notice that the coefficients for JavaChat and Lampiro are below of the mean. This happens because of the relative low number of feature dependencies.

5. Discussion and Implications

This section performs exploratory analyses of the results presented in Section 4 to explain the relationship between feature dependencies and change propagation. Section 5.1 analyses the

collected data for answering RQ1, i.e. the link between feature dependency and simultaneous change. Addressing RQ2, Section 5.2 presents a discussion about the impact of feature dependencies on change propagation and their implications. Finally, Section 5.3 discusses the inequality of the distribution of change propagation through paths of feature dependencies in order to understand the answer of RQ3.

5.1. The Relation between Feature Dependency and Change Propagation

The pronounced difference between P_{FD} and $P_{\neg FD}$ provides evidence that the existence of a dependency raises the chance of a change propagation between features. In addition, the difference between P_C and $P_{\neg C}$ shows us that simultaneous changes are more likely to happen in dependent features than in independent features. So, coming back to RQ1, the answer is *yes*. Moreover, we can conclude that the results support our argument that dependencies are closely related to change propagation.

Buried feature interfaces. To understand why feature dependencies are likely to propagate changes, we analysed the source code of the SPLs. In our analysis, we noticed that developers often change parts of the code that affect program elements responsible for realising the communication between features. As a consequence, these actions demand changes on dependent features – i.e. a change propagation. For instance, in the example presented in Section 2, the developer changed an attribute read by the method `getWeight()`, which is a member of the interface of the feature WEIGHTED. This change was propagated to feature DIJKSTRA due to its dependency on feature WEIGHTED. In this context, we can argue about the importance of identifying, buried into hundreds of lines of code, the program elements responsible for realising feature dependencies when maintaining features. These program elements might be considered as members of a conceptual feature interface. So, having explicit the members of a conceptual feature interface would indicate the critical parts of the code that must be carefully changed due to a high probability of change propagation.

Alternative features. Another interesting point in our data is related to a specific case of simultaneous changes that happen in independent features ($P_{\neg FD}$). There were recurrent cases where alternative features simultaneously changed during an evolution. Alternative features are features that are mutually exclusive in a product generated by the SPL. The way this type of constraint was implemented in our target SPLs (and in most preprocessor-based SPLs) is not considered as a dependency in our study (see Section 3.2 for the cases of dependencies we are considering). These constraints usually are implemented by means of preprocessor directives. However, this mutually exclusive relationship between these features may have an impact on change propagations. Figure 12 illustrates this situation considering a scenario with three alternative features in the example presented in Section 2.

In Figure 12, we can notice that the inclusion of feature BELLMAN-FORD causes simultaneous changes in three features: the feature ALGORITHMS and the alternative features

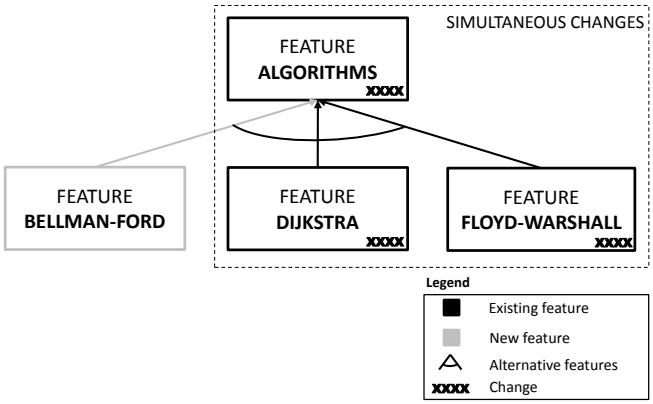

Figure 12: Simultaneous changes in alternative features and in the feature ALGORITHMS.

DIJKSTRA and FLOYD-WARSHALL. There is no dependency in the source code between DIJKSTRA and FLOYD-WARSHALL, but there are simultaneous changes in these features due to SPL domain constraints involving them. A change in program elements of the feature ALGORITHMS, that are responsible for the communication between this features with other features (i.e. members of a feature interface), is the cause of simultaneous changes in alternative features.

Alternative features often use the same members of a conceptual feature interface. So, a change that affects these members will probably affect all alternative features using the same feature interface members, thus causing simultaneous changes in independent features. This means that feature interface members should be carefully maintained due to their impact on other features, mainly when it involves alternative features. To ameliorate this type of situation, we reinforce that explicit all the members of a feature interface must help developers to reason about an evolution considering the dependencies between features.

Feature dependencies are not alike. A cross-reading of the calculated probabilities shows us that the values of P_{FD} and $P_{\neg FD}$ are inversely proportional to P_C and $P_{\neg C}$, respectively. For instance, the lowest values of P_{FD} and $P_{\neg FD}$ from BerkeleyDB contrast with its highest values of P_C and $P_{\neg C}$. The low values of P_{FD} and $P_{\neg FD}$ in BerkeleyDB are due to the high number of feature dependencies (860 dependencies) and due to the number of change propagation happening in a few dependencies. These low values could lead us question the validity of our answers regarding RQ1 that state that there is a relation between feature dependency and change propagation. However, the high values of P_C and $P_{\neg C}$ confirm our statement by showing that when there is a simultaneous change between features, there is a high probability of having a dependency amongst these features. Thus, this behaviour in BerkeleyDB, which is not an exception amongst the the SPL analysed, can be interpreted as an indication of concentration of change propagation in certain feature dependencies. A few number of feature dependencies seems to concentrate a high number of recurrent simultaneous change, thus indicating that there are feature dependencies that are more likely to propagate changes. This impor-

tant point will be revisited in the discussion on inequality of the distribution of changes through the dependencies (Section 5.3) addressed by RQ3.

5.2. Extent of the Change Propagation

The results presented in the previous section (Section 5.1) indicate that feature dependencies are related to change propagation. In our RQ2, we were interested in discovering the extent of this propagation through paths of dependencies. The results presented in Section 4.2 show us the relation between probability of change propagation and distance between dependent features. Analysing the collected data, we found that there is a decreasing probability of change propagation as the distance between dependent features increases in the path of dependencies. Moreover, we observed that this relationship can be approximated as a linear function. So, these results evidence the transitive propagation of changes via a path of dependencies.

This result about the relationship between distance and change propagation may have important implications. A property that shows the *depth of the dependency* would indicate the axes of change in a SPL. So, features involved in these axes of change are likely to suffer changes related to change propagation if the evolution affects one of these features. Thus, developers should be concerned with these specific features during the SPL evolution based on the property *depth of the dependency*. Reasoning about this property might help decreasing the maintenance effort of SPLs along the evolutions. In addition, the data show a linear decreasing relationship between distance and probability of change propagation indicating a more severe change propagation in SPL features than in classes. For instance, Geipel and Schweitzer [8] found an exponential decay regarding the relation between probability of changes and distance amongst classes. In other words, it is more likely the extent of the change propagation in SPLs reach more modules (features) than the change propagation in modules of stand-alone programs (classes). Therefore, the *depth of the dependency* might help developers to concern about paths of feature dependencies during the maintenance of a feature in order to minimise the change propagation.

Another point in our data is related to the depth of feature dependencies that occur in SPLs. Most of the SPL releases (77%) have a *depth of the dependency* higher than one. This information might be valuable to tune and/or complement existing approaches, such as combinatorial interaction testing. For instance, there are several sampling-based analysis techniques in combinatorial interaction testing that aim at covering all pairs of feature (i.e. *depth of the dependency* equal to 1), but disregard larger feature combinations [41, 42, 43]. So, these approaches assume that a major fraction of feature dependencies have a *depth of the dependency* equal to one. According to our results, this assumption might have implications for the precision of those approaches. Since dependencies can propagate changes, it is possible that these dependencies can also propagate errors. In this case, approaches covering only a *depth of the dependency* equal to one might be missing information to reveal errors, change propagation estimation, and the like.

5.3. Consequences of the Distribution Inequality

The results presented in this paper show a strong relation between feature dependency and change propagation. Moreover, we provide evidence for the propagation of changes via dependencies. Apart from these points, the data obtained for answering RQ3 challenges the common assumptions on the relationship between dependencies and change propagation. Some studies [44, 45], which evaluate non-SPLs, implicitly assume an equal distribution of change propagation amongst dependencies. In other words, the dependency structure would be a measure for the software change behaviour in these studies. However, our data show an inequality in the distribution of change propagation through the SPL dependencies. So, it might be problematic to simply adapting stand-alone software approaches to analyse the change behaviour of a SPL.

The inequality in the distribution of change propagation indicates that we cannot treat all dependencies alike. Few dependencies are related to most change propagation of a SPL evolution history. Based on this, we can pinpoint two important findings. First, characterising properties based on the source code is needed in order to differentiate feature dependencies. In this direction, there are other studies that identify characterising properties for feature dependencies, such as volatility and scope [12, 18]. Volatility refers to the extent that dependencies between program elements may be broken when a single change is performed. Scope refers to the program elements involved in the dependency. For instance, when a feature dependency has a high scope, many program elements are involved in establishing this feature dependency. When a change affects one of the many program elements, it is likely that the change must be propagated to dependent features. So, a high scope in feature dependencies may indicate that such dependencies are more likely to concentrate change propagations. In our study, dependencies that concentrated higher numbers of change propagation presented a low volatility and a high scope. So, these and other properties (e.g. depth of feature dependency - Section 5.2) may indicate which dependencies must gain more attention during an evolution in order to reduce the change propagation.

Second, as many dependencies are not involved in change propagation, only reducing all the dependencies will not necessarily reduce change propagation in SPLs. Furthermore, dependencies might indicate a high reuse [8]. In other words, a dependency indicate that a functionality was not duplicated but reused. So, a highly connected feature is not necessarily an indicator of flawed design, but it might indicate a key feature to the SPL architecture. In addition, since these features are highly connected, it is natural they present a large (and not cohesive) feature interface. Therefore, we argue that, besides making feature interfaces explicit, grouping the members of the feature interface would support developers to predict changes that happen in the presence of feature dependencies and drive efforts to specific parts of the source code guided by the interface.

6. Study Limitations

This section discusses the study limitations based on the four categories of validity threats described by Wohlin et al. [46]. Each category has a set of possible threats to the validity of an experiment. We identified these possible threats to our study within each category, which are discussed in the following with the measures we took to reduce each risk.

Conclusion Validity. The major risk here is related to the *random heterogeneity of subjects:* the chosen SPLs came from different application domains (Section 3). In other words, 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 promote the external validity of the study by improving the ability to generalise the results of our experiment.

Internal Validity. The detected risk is that we are considering the data from multiple releases all together in our matrices of feature dependencies and simultaneous change. In other words, we are not considering the effect of time when analysing the concentration of change propagation on feature dependencies. Despite of the aggregated data, we argue that this risk is minimised due to the simulation of the evolution scenarios as well the number of evolutions. Although simulating the evolution scenarios does not mean controlling the change propagation, we argue that we minimise the effect of time due to the diversity of the evolution scenarios. In other words, we minimise the chances of change propagations happen in feature dependencies that exist since early evolutions by controlling the evolution scenarios.

Construct Validity. We detect two possible threats related to the *restricted generalizability across constructs*: (i) the use of conditional compilation as the variability mechanism for implementing features in the source code might increase the number of feature dependencies when compared to other variability mechanisms. With conditional compilation, features may be scattered in the source code. This means that a feature may present low cohesion, and, as consequence, a high coupling with other features (i.e. high number of feature dependencies), and (ii) we focus only in evolution with perfective maintenance, i.e. mainly the addition of new features. In this case, we might have other maintenance scenarios where the results are not similar to the ones presented in this paper. Risk (i) cannot be completely avoided as all SPLs analysed are implemented using the mechanism of conditional compilation. However, we argue that conditional compilation is the most widely used mechanism to implement SPL features [17]. Moreover, except for the Graph Product Line, all target SPLs are non-academic projects. So, we believe these SPLs were developed aiming for a good design, thus reducing the number of unnecessary feature dependencies. Nevertheless, the Graph Product Line was continuously improved over years by an academic community that uses it as a subject of studies. Risk (ii) also cannot be avoided due to the design of the study. However, we argue that more than 60% of the maintenance tasks are associated to a perfective maintenance [36]. In other words, it is the most common maintenance type in software evolution.

External Validity. We identified two risks in this category related to the interaction of setting and treatment: (i) the target SPLs might not be representative of the industrial practice, and (ii) the evolution scenarios might not represent relevant scenarios of evolution. We also identified one risk related to the interaction between selection and treatment: (iii) the subject responsible for implementing the evolution scenarios might not be representative of the population we want to generalise to. In order to reduce risk (i), we evaluated SPLs that come from heterogeneous application domains. In addition, all SPLs have been extensively used and evaluated in previous research [9, 47, 48]. We believe the characteristics of the selected SPLs, when contrasted with the state of practice in SPL, represent a first step towards the generalisation of the results achieved in this study. Regarding risk (ii), we created SPL evolution scenarios based on studies of SPL evolution [22, 34, 35]. Moreover, we defined clear procedures for the creation of each SPL release. Design practices were used, enforced, and reviewed throughout the creation of evolution of all the SPL releases. In all the cases, the evolution scenarios were also reviewed by experts in the field. Finally, although this risk is considered a threat to the external validity, it also helps to promote the construct validity of the study. Regarding risk (iii), similar to risk (ii), we argue that we defined clear procedures for the evolution, design practices were used, and the evolution scenarios were reviewed by experts in the field.

7. Related Work

Software change and change propagation. Many research work have explored the understanding of software change and the impact of those changes on specific software properties [8, 21, 44, 45]. Most of these investigations, however, only concentrate on the analysis of module dependencies in standalone systems. For instance, Geipel and Schweitzer [8] investigate the relationship between class dependency and change propagation. The study concludes a strong relationship between dependency and change propagation. Moreover, they revealed half of all dependencies are never involved in change propagation. In a study about the impact of changes, Sangal et al. [45] propose an approach that uses dependency models in order to manage complex software architecture. Our work deals with two different main aspects when compared to those related work. First, we focus our research on feature change. SPLs use features as the unit of abstraction. Thus, we explore changes on SPL features instead of the unit of abstraction of programming languages used to implement SPLs (e.g.: classes or aspects). Second, we consider only SPL in our study. A SPL allows the generation of several products by combining different SPL's features. So, the extent of a change propagation may affect from dozens to thousands different products depending on the characteristics of the SPL.

Feature dependencies vs. SPL maintenance. There are also investigations trying to understand and minimise negative effects of feature dependencies on SPL development. Cataldo and Herbsleb [10] have empirically studied feature-oriented development in order to observe the impact that some

attributes of this type of development have on integration failures. They concluded that dependencies and cross-feature interactions are drivers of integration failures. In another work related to feature dependencies, Ribeiro et al. [9] performed an empirical evaluation on forty preprocessor-based SPLs focusing on the maintainability of feature dependency code. They proposed an approach to reduce the effort of maintenance in SPL implemented using preprocessor by focusing on feature dependencies. The conclusion was that feature dependencies are reasonably common in preprocessor-based SPLs. Moreover, they highlight the impact of feature dependency on the maintenance by increasing the maintenance effort. These and other authors [11, 12, 18, 48, 49, 50], have been highlighted the impact of feature dependencies on several attributes of SPLs. For instance, Cafeo et al. [12] explore the influence of different programming techniques on feature dependency implementation regarding change propagation. In [18], Cafeo et al. explored the power of feature dependencies in indicate change propagation. Finally, in [51], Cafeo et al. presented initial evidences that feature dependencies may be a main driver to change propagation. Although related, [12] and [18] are complimentary to this work, since they explore the relation studied in this study. Moreover, we evolve and explore the idea presented in [51] in this paper, and we try to understand the link between feature dependency and change propagation because (i) there is no empirical evidence about the relation between change propagation and feature dependency, and (ii) change propagation may impact on several other SPL attributes such as failures and maintenance effort.

8. Conclusions

This paper presented an exploratory study on perfective maintenance of SPLs implemented with conditional compilation. We analysed five SPLs from different application domains, from which we collected (i) simultaneous changes in features during the evolution, and (ii) feature dependencies upon which we performed our analyses. We built our evaluation procedure on work by Geipel and Schweitzer [8] which we adapt to the context of SPLs and features. The contribution of this paper is having empirically analysed the relation between feature dependency and change propagation. Moreover, this analysis allowed us to conclude about the causes and consequences of change propagation along feature dependencies of SPLs.

The results revealed a relation between feature dependency and change propagation. Basically, features that present a dependency are more likely to change together during an evolution than independent features. Moreover, we identified parts of the source code that are possible causes of this strong relation. Our analysis also evidenced the relation between change propagation and distance amongst features in the dependency network. Such analysis pointed to a linear relation between distance and change propagation, which indicates a more powerful change propagation in features than the exponential relation between distance and classes [8]. Finally, the results also revealed a inequality in the distribution of change propagation through the feature dependencies of a SPL. This counterintuitive result indicates that (i) a general feature dependency minimisation might not ameliorate the change propagation; and (ii) characterising feature dependency properties must be analysed to identify the axes of changes in feature dependencies.

We believe that the results presented in this paper improve the understanding of the behaviour of feature dependencies regarding change propagation, motivate more intensive research on change propagation in SPLs as well as in feature dependency characterising properties, motivate studies in different mechanisms and languages, and provide directions for enhancing automating SPL evolution.

Acknowledgements

This work has been funded by: B. B. P. Cafeo CAPES PhD scholarship and CNPq scholarship (number 141688/2013-0); A. Garcia FAPERJ distinguished scientist grant (number E-26/102.211/2009), CNPq productivity grant (number 305526/2009-0 and 308490/2012-6), Universal project grants (number 483882/2009-7 and 485348/2011-0), and PUC-Rio (productivity grant).

References

[1] P. Clements, C. Kreuger, Point/counterpoint, IEEE Software 19 (4) (2002) 28–31.
[2] S. Apel et al., Strategies for product-line verification: case studies and experiments, in: ICSE 2013, IEEE Press, 2013, pp. 482–491.
[3] K. Pohl et al., Software Product Line Engineering: Foundations, Principles and Techniques, Springer-Verlag New York, Inc., 2005.
[4] M. M. Lehman, L. A. Belady, Program evolution: processes of software change, Academic Press, Inc., 1985.
[5] D. L. Parnas, Software aging, in: ICSE 1994, IEEE Computer Society Press, 1994, pp. 279–287.
[6] H. Ye, H. Liu, Approach to modelling feature variability and dependencies in software product lines, IEEE Software 152 (3) (2005) 101–109.
[7] V. Rajlich, A model for change propagation based on graph rewriting, in: ICSM 1997, IEEE Computer Society, 1997, pp. 84–91.
[8] M. M. Geipel, F. Schweitzer, The link between dependency and cochange: Empirical evidence, IEEE Transactions on Software Engineering 38 (6) (2012) 1432–1444.
[9] M. Ribeiro et al., On the impact of feature dependencies when maintaining preprocessor-based software product lines, in: GPCE 2011, ACM, 2011, pp. 23–32.
[10] M. Cataldo, J. Herbsleb, Factors leading to integration failures in global feature-oriented development: an empirical analysis, in: ICSE 2011, 2011, pp. 161–170.
[11] B. Garvin, M. Cohen, Feature interaction faults revisited: An exploratory study, in: ISSRE 2011, IEEE Computer Society, 2011, pp. 90–99.
[12] B. B. P. Cafeo et al., Analysing the impact of feature dependency implementation on product line stability: An exploratory study, in: SBES 2012, IEEE, 2012, pp. 141–150.
[13] M. Godfrey, D. German, The past, present, and future of software evolution, in: FoSM 2008, 2008, pp. 129–138.
[14] P. C. Clements, L. Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley, 2001.
[15] S. Trujillo, D. Batory, O. Diaz, Feature refactoring a multi-representation program into a product line, in: GPCE 2006, ACM, New York, USA, 2006, pp. 191–200.
[16] S. Apel et al., Feature-Oriented Software Product Lines, Springer-Verlag New York, Inc., 2013.
[17] C. Kästner, S. Apel, Virtual separation of concerns – a second chance for preprocessors, Journal of Object Technology 8 (6) (2009) 59–78.

[18] B. B. P. Cafeo et al., Towards indicators of instabilities in software product lines: An empirical evaluation of metrics, in: WETSoM 2013, IEEE Press, 2013, pp. 69–75.
[19] E. Dijkstra, A note on two problems in connexion with graphs, Numerische Mathematik 1 (1959) 269–271.
[20] R. Bellman, On a Routing Problem, Quarterly of Applied Mathematics 16 (1958) 87–90.
[21] A. E. Hassan, R. C. Holt, Predicting change propagation in software systems, in: ICSM 2004, IEEE Computer Society, 2004, pp. 284–293.
[22] E. Figueiredo et al., Evolving software product lines with aspects: an empirical study on design stability, in: ICSE 2008, ACM, 2008, pp. 261–270.
[23] R. W. Floyd, Algorithm 97: Shortest path, Communications of the ACM 5 (1962) 345–.
[24] C. Kästner, CIDE tool (January 2014).
URL http://wwwiti.cs.uni-magdeburg.de/iti_db/research/cide/
[25] C. Kästner, S. Trujillo, S. Apel, Visualizing software product line variabilities in source code, in: ViSPLE 2008, 2008, pp. 303–312.
[26] E. Cirilo, U. Kulesza, C. J. Lucena, A product derivation tool based on model-driven techniques and annotations, Journal of Universal Computer Science 14 (8) (2008) 1344–1367.
[27] B. B. P. Cafeo et al, Feature dependencies as change propagators: An exploratory study of software product lines (January 2014).
URL http://www.inf.puc-rio.br/~bcafeo/supsite_ist2015.html
[28] Oracle, Lampiro (September 2014).
URL http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/downloads/index.html
[29] T. Thüm, F. Benduhn, SPL2Go (August 2013).
URL http://spl2go.cs.ovgu.de/
[30] Mobile RSS, Mobile rss (September 2014).
URL https://code.google.com/p/mobile-rss/
[31] Lampiro, Lampiro (September 2014).
URL https://code.google.com/p/lampiro/
[32] XMPP Standards Foundation, Xmpp (August 2013).
URL http://xmpp.org/xmpp-protocols/
[33] R. Lopez-Herrejon, D. Batory, A standard problem for evaluating product-line methodologies, in: GCSE 2001, Springer-Verlag, 2001, pp. 10–24.
[34] V. Alves et al., Extracting and evolving mobile games product lines, in: SPLC 2005, Springer, 2005, pp. 70–81.
[35] M. Svahnberg, J. Bosch, Evolution in software product lines, in: IWSAPF, Springer LNCS, 2000, pp. 391–422.
[36] S. R. Schach et al., Determining the distribution of maintenance categories: Survey versus measurement, Empirical Software Engineering 8 (4) (2003) 351–365.
[37] D. P. Bertsekas, J. N. Tsitsiklis, Introduction to Probability, 2nd Edition, 2nd Edition, Athena Scientific, 2008.
[38] B. Vasilescu, A. Serebrenik, M. Van Den Brand, You can't control the unfamiliar: A study on the relations between aggregation techniques for software metrics, in: ICSM 2011, IEEE, 2011, pp. 313–322.
[39] M. O. Lorenz, Methods of Measuring the Concentration of Wealth, Publications of the American Statistical Association 9 (70) (1905) 209–219.
[40] C. Gini, Measurement of inequality of incomes, The Economic Journal 31 (1921) 124–126.
[41] D. Marijan et al., Practical pairwise testing for software product lines, in: SPLC 2013, ACM, 2013, pp. 227–235.
[42] G. Perrouin et al., Pairwise testing for software product lines: comparison of two approaches, Software Quality Journal 20 (3-4) (2012) 605–643.
[43] M. Lochau et al., Model-based pairwise testing for feature interaction coverage in software product line engineering, Software Quality Journal 20 (3-4) (2012) 567–604.
[44] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, Predicting the probability of change in object-oriented systems, IEEE Transactions on Software Engineering 31 (7) (2005) 601–614.
[45] N. Sangal et al., Using dependency models to manage complex software architecture, in: OOPSLA 2005, ACM, 2005, pp. 167–176.
[46] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering: An Introduction, Kluwer Academic Publishers, 2000.
[47] J. Liebig et al., An analysis of the variability in forty preprocessor-based software product lines, in: ICSE 2010, ACM, 2010, pp. 105–114.
[48] S. Apel, D. Beyer, Feature cohesion in software product lines: an exploratory study, in: ICSE 2011, ACM, 2011, pp. 421–430.
[49] S. Ferber et al., Feature interaction and dependencies: Modeling features for reengineering a legacy product line, in: Software Product Lines, Vol. 2379 of LNCS, Springer Berlin Heidelberg, 2002, pp. 235–256.
[50] K. Lee, K. C. Kang, Feature dependency analysis for product line component design, in: Software Reuse: Methods, Techniques, and Tools, Vol. 3107 of LNCS, Springer Berlin Heidelberg, 2004, pp. 69–85.
[51] B. Cafeo et al., On the relation between feature dependencies and change propagation, in: Feature Interactions: The Next Generation (Dagstuhl Seminar 14281), Vol. 4 of Dagstuhl Reports, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014, pp. 18–18.