Segregating Feature Interfaces to Support Software Product Line Maintenance

Bruno B. P. Cafeo
Opus Research Group, PUC-Rio, Brazil
bcafeo@inf.puc-rio.br

Claus Hunsen
University of Passau, Germany
hunsen@fim.uni-passau.de

Alessandro Garcia
Opus Research Group, PUC-Rio, Brazil
afgarcia@inf.puc-rio.br

Sven Apel
University of Passau, Germany
apel@uni-passau.de

Jaejoon Lee
Lancaster University, UK
j.lee3@lancaster.ac.uk

Abstract

Although software product lines are widely used in practice, their maintenance is challenging. Features as units of behaviour can be heavily scattered across the source code of a product line, hindering modular reasoning. To alleviate this problem, feature interfaces aim at enhancing modular reasoning about features. However, considering all members of a feature interface is often cumbersome, especially due to the large number of members arising in practice. To address this problem, we present an approach to group members of a feature interface based on their mutual dependencies. We argue that often only a subset of all interface members is relevant to a maintenance task. Therefore, we propose a graph representation that is able to capture the collaboration between members and apply a clustering algorithm to it to group highly-related members and segregate non-related members. On a set of ten versions of a real-world product line, we evaluate the effectiveness of our approach, by comparing the two types of feature interfaces (segregated vs. original interfaces) with co-change information from the version-control system. We found a potential reduction of 62% of the interface members to be considered during maintenance.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques—Modules and Interfaces; K.6.3 [*Software Engineering*]: Software Management —Software Maintenance

Keywords Software Product Lines, Feature Interface, Feature Dependencies.

1. Introduction

Software product lines have gained considerable momentum in recent years, both in industry and in academia [4]. Product-line engineering facilitates software reuse by decomposing software into units of functionality, called features [4]. *Features* are cohesive units of behaviour and used to express commonalities and variabilities of the products of a product line [7]. Features often need to cooperate with other features to fulfil specific tasks [4], inducing feature dependencies. A *feature dependency* manifests in the source code of a product line whenever one or more program elements (e.g., code blocks, methods, or fields) inside the boundaries of a feature depend on elements external to that feature [33]. A simple example is a variable defined in one feature and used in another feature.

The maintenance of industry-strength product lines is particularly hard due to the lack of feature modularity [10, 34]. First, features are often scattered across the source code [18]. Hence, product-line developers are forced to consider all scattered code to reason about a given feature during a maintenance task. This problem is exacerbated in preprocessor-based software lines, which are very common in the software industry [16]. Second, and more importantly, the communication between a feature with other features is often realised by many code-level dependencies also scattered across the source code [33]. As a consequence, a lot of effort spent in product-line maintenance is devoted to identifying and understanding program elements responsible for giving external access to other features as well as to reading code of potentially affected features [33].

Developers cope with the complexity of maintaining software systems by reasoning about interfaces [30]. The benefits of identifying and structuring interfaces in software systems have been a major focus of studies in the 80s [29, 30, 35], but there is a also growing interest in understanding how to tame the complexity of so-called feature interfaces in software product lines [19, 22, 34]. In the context of software product lines, the *interface of a feature* contains the program elements in the source code that are responsible for providing external access to other features [10, 33]. In particular, researchers have found that explicitly reasoning about feature interfaces reduces the effort of developers to maintain the system [34]. However, only identifying and making feature-interface members explicit may be insufficient for enhancing modular reasoning of features [10]. First of all, feature interfaces may have many members responsible for the communication between features. As a consequence, feature interfaces may be large and not helpful for supporting developers during maintenance. Furthermore, according to previous experience [10, 36], among the many members of a feature interface, only a small group of members may be relevant to a given maintenance task—as it contains a set of highly related members always used together, for example. So, a simple list of all interface members might rather hinder than improve modular reasoning of features. As an extension to existing studies, we investigate how to support developers in identifying and structuring feature interfaces.

To tackle this problem, we propose a technique for automating the segregation of feature interfaces (i.e., grouping of interface members) based on their mutual dependencies. The background is that developers cope with the complexity of large software systems by grouping (clustering) related members into cohesive groups [29]. In object-oriented design, for instance, interface members are segregated into more cohesive groups according to their clients [29]. In the same vein, feature interfaces shall be segregated into cohesive clusters of members, such that the members of a cluster collaborate closely to accomplish a part of the overall purpose of a feature in cooperation with another feature (i.e., feature dependency). If a change must be made in a member of a cluster, the members of the same cluster are likely highly relevant to be revised during maintenance. In other words, only a subset of all interface members may be relevant to a maintenance task.

To achieve this goal, we formulate interface segregation as a clustering problem and conduct a study on an industrial product line. While using maintenance tasks (i.e., inferred from commits) as a foundation, we analyse the number of interface members likely to be unnecessarily considered by developers to the clusters of the segregated feature interface, on the one hand, and to the original interface (i.e., one subset of all interface members), on the other hand. To this end, we propose a graph representation to capture the collaboration between interface members, and we apply a clustering algorithm on this representation to identify the clusters of members, containing only members that are highly related. The results show a pronounced difference ($\approx 62\%$) in favour of segregated interfaces regarding the reduction of interface members likely unnecessarily considered by developers—i.e., members not changed in a commit—during maintenance. Therefore, by capturing cooperating interface members and clustering them, we are able to reduce the overall amount of code considered by developers during a maintenance task. In summary, our contributions are:

- A graph representation for relating feature-interface members.
- A tool that is able to generate segregated feature interfaces, and the use of this tool to generate the segregated interfaces for all features of 10 versions of BusyBox.
- Evaluation of our approach by comparing the original feature interfaces with the segregated interfaces by means of the Jaccard distance (using the co-changed interface members as oracle); we found that segregated interfaces potentially reduce the overall amount of code considered by developers in product-line maintenance.

2. Preliminaries

To lay a foundation for the subsequent sections, we introduce the relevant concepts of software product lines, features, and feature dependencies. Furthermore, we introduce a motivating example to illustrate the problem of understanding feature dependencies by means of feature interfaces.

2.1 Software Product Lines and Features

A *software product line* 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" [11]. Software product lines enable the systematic construction of individual software products via mass customisation. Customers tailor their products by selecting particular combinations of product-line features. The use of software product lines promises significant benefits, such as a reduction of development costs, enhancement of quality, and reduction of time-to-market [11]. To this end, product lines are organised and structured in terms of features. *Features* are units of behaviour of by which different products within a product line can be differentiated and defined [4, 37], thus, playing a key role for mass customisation.

In this work, we look at product lines implemented with a preprocessor and conditional compilation, which is a widely-used approach to implement and configure product-line features [16]. The preprocessor identifies the code that should be compiled or not based on preprocessor directives and propositional expressions over features they contain, called the presence conditions. In this setting, a feature is a set of program elements surrounded by preprocessor directives using the same presence condition. It is important to notice that features might be scattered accross several modules (e.g., compilation units) of the source code and tangled with other feature code. Hence, the maintenance of product-line features is often more challenging than the maintenance of modules.

As an example, we will use a simple product line for managing devices (such as printers, displays, and the like), implemented using preprocessor directives and conditional compilation. The user is able to control devices, either via the internal screen or externally via a terminal connection. Figure 1 illustrates a code snippet of how a feature might be represented in the source code in our exemplary product line, specifically, parts of the feature SCREEN implementing the basic internal screen functionality. In this snippet, we show the declaration of the fields responsible for the width and height of the screen (Line 3), and the methods responsible for setting the values of these fields (Lines 7–8).

```
 1  public class General {
 2    #ifdef SCREEN
 3      int width; int height;
 4    #endif
 5    ...
 6    #ifdef SCREEN
 7      public void setWidth(int x) {this.width = x;}
 8      public void setHeight(int y) {this.height = y;}
 9    #endif
10  }
```

Figure 1. Excerpt of the code of feature SCREEN.

2.2 Feature Dependencies and Feature Interfaces

A *feature dependency* defines which program elements of a feature depend on other elements of other features (e.g., a method is defined in one feature and called by another feature) [10]. In other words, feature dependencies arise, among others, from structural dependencies (such as feature control-flow dependencies and inheritance calls) between program elements of different features.

Figure 2 illustrates an example of a feature dependency between the feature ADMINPANEL and SCREEN in our example. ADMINPANEL is responsible for setting values to devices. The feature dependency arises from the method calls highlighted in Lines 4, 5, 6, 7, 11, and 12, while some of these methods belong to feature SCREEN (Lines 9–14, Figure 1). Thus, a part of the ADMINPANEL feature depends on elements of feature SCREEN. Furthermore, we have a feature INFOPANEL responsible for presenting the current settings to the user (no code snippet given), which accesses the methods `getFrequency` and `getResolution`, among others, that just read the values configured.

A *feature interface* consists of the program elements that are responsible for providing external access to other features. A *provided* feature interface, similarly to a provided module interface, consists of the program elements belonging to a feature and used by another feature. We focus on provided feature interfaces as required feature interfaces can be inferred once the former ones have been identified. So, from hereafter, unless otherwise stated, the term "feature interface" is used to refer to the provided feature interface. The methods called in Lines 5, 6, 7, 8, 16, and 17 of Figure 2 are the elements responsible for providing external access to feature SCREEN. Therefore, the methods `setWidth`, `setHeight`,

```
 1  public class Controller {
 2    #ifdef ADMINPANEL
 3      public void resetConfig() {
 4        scr.setWidth(1920);
 5        scr.setHeight(1080);
 6        scr.setBrightness(80);
 7        scr.setContrast(30); ...
 8      } ...
 9      public void correctAspectRatio(int x, int y) {
10        // adjust aspect ratio if necessary
11        scr.setWidth(x*factorW);
12        scr.setHeight(y*factorH); ...
13      }
14    #endif
15  }
```

Figure 2. Excerpt of the code of feature ADMINPANEL, giving rise to a feature dependency (underlined).

`setBrightness`, and `setContrast` are members of the feature interface SCREEN.

2.3 A Maintenance Problem

We now introduce an example of a maintenance problem to illustrate the need for feature interfaces. In our exemplary product line, there are two sets of functions belonging to feature SCREEN that participate in feature dependencies: (i) the functions `showFrequency` and `showResolution` are responsible for external access to feature SCREEN by providing necessary information, and (ii) the functions `setWidth`, `setHeight`, `setBrightness`, and `setContrast` allow the user to set configuration parameters. These elements, all belonging to the feature SCREEN, are scattered across several files of the code base, as shown in Figure 3.

Let us suppose a maintenance task where the developer is going to limit the size of the screen to values between a minimum and maximum possible size. So, it is expected a change in feature SCREEN to accomplish the maintenance task. However, since features the ADMINPANEL and INFOPANEL depend on feature SCREEN, the developer should revisit the code of all these dependent features to ensure that their expectations are met. Figure 4 illustrates the interface of feature SCREEN.

Making feature-interface members explicit would help spotting feature dependencies. For instance, developers can associate members starting with the prefix `set` with the feature ADMINPANEL. However, despite the seeming benefit, reasoning solely about a list of interface members—the so-called original feature interface—may be still too complicated. Original feature interfaces may be large and not helpful for supporting developers during maintenance. A simple list

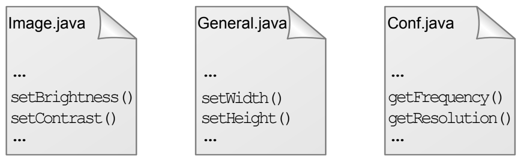

Figure 3. Methods of feature SCREEN involved in feature dependencies.

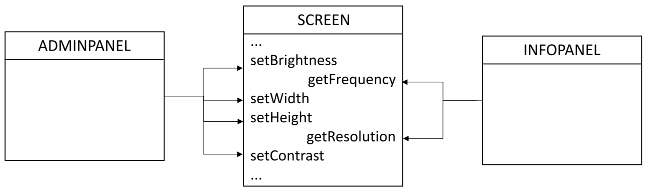

Figure 4. Features ADMINPANEL and INFOPANEL accessing methods of feature SCREEN.

of all interface members might rather hinder than enhance modular reasoning of features. In the subject system of our study (Section 4.2), for instance, we found interfaces with more than 20 members. In this case, analysing each feature interface member to reason about feature dependencies during a maintenance task may be even worse than not having a explicit feature interface.

In the context of our example, the previously-mentioned maintenance task would touch the interface members `setWidth` and `setHeight` of feature SCREEN. In addition, it may also demand changes to the code of feature ADMINPANEL, which is referring to `setWidth` and `setHeight`. On the other hand, the other interface members of feature SCREEN, other parts of feature ADMINPANEL, and the code of feature INFOPANEL may not be important to be considered in this specific maintenance task. Therefore, the key idea is to group members of a feature interface that are closely related, which may help developers to focus only on important members of the feature interface relevant to the maintenance task. This way, the lack of modular reasoning of features could be reduced as well as the maintenance-task difficulty.

3. Automated Interface Segregation

Creating a human-perceivable model of the structure of a complex system is one of the many problems of software engineering [27]. In our attempt to alleviate the complexity of reasoning about features as modular units of behaviour, we propose a technique for automated feature-interface segregation to support developers creating a more understandable model of the program structure in terms of features and their dependencies. The goal of our approach is to automatically partition[1] the members of a feature interface into clusters based on their dependencies to other features. By doing that, the resulting clusters increase the chance of interface members of the same cluster being important to be understood together. The clusters, once discovered, will represent a higher-level abstraction of a feature interface based on the global feature-dependency structure. Each cluster contains a set of interface members that cooperate to perform a high-level function together. The following sections detail our approach to organise feature interfaces using clustering.

[1] We use the term *partition* in the traditional mathematical sense, that is, the decomposition of a set of elements (e.g., nodes of a graph) into mutually disjoint clusters.

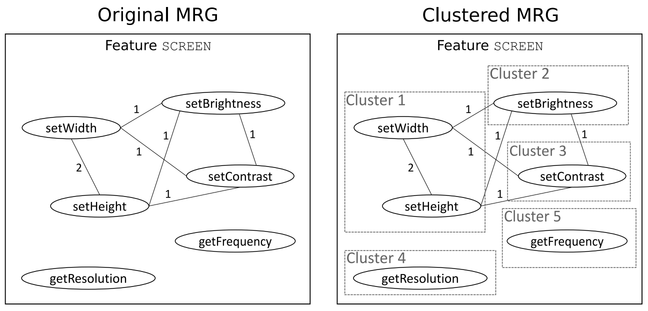

Figure 5. Example of an MRG of feature SCREEN before and after clustering.

3.1 Interface Organisation as a Clustering Problem

Clustering is the task of grouping a set of objects, such that objects in the same group (i.e., on the same cluster) are more similar (in some sense or another) to each other than to those in other groups (clusters). Clustering is widely used in many applications, such as data mining, Web analysis, and computational biology [13].

By grouping related members of a feature interface together—i.e., members that cooperate to perform a high-level function together—developers should be pointed to members that actually correspond to the current maintenance task. This way, the effort to comprehend dependencies could be reduced, and thus the maintenance-task difficulty and number of potentially related program elements to observe.

To cluster feature interfaces, we exploit the relationships between interface members and apply a clustering algorithm to partition the members of feature interfaces according to our approach. In Section 3.2, we define the graph representation used to capture those relationships.

3.2 Member Relationship Graph

To cluster the members of a feature interface, the straightforward representation is a graph, which we call *Member Relationship Graph* (MRG). Formally, an $MRG = (M, R)$ consists of two components M and R, where M is the set of members in a specific feature interface, and $R \subseteq M \times M$ is a set of pairs of the form $\langle u, v \rangle$ which represents the members' relationships. Figure 5 illustrates the idea of the MRG by means of our example.

A relationship between two feature interface members of a single feature exists when the same program element of a dependent feature refers to both feature interface members. For instance, based on the example presented in Section 2.3, the interface members `setWidth` and `setHeight` of feature SCREEN are referred to by two program elements (`resetConfig` and `correctAspectRatio`). The rationale behind this graph structure is that interface members referred to by the same program element from another feature are likely to cooperate to perform part of the functionality the other feature needs. As a consequence, they likely give together a more complete insight into the corresponding feature dependency. In addition, each edge of the graph has a weight.

The weight of an edge represents the number of distinct program elements referring to the pair of members. The more the pair is used together the more related they are in the source code. An edge between two vertices with a high weight means the elements should be in the same cluster. In our example presented in Section 2.3, the interface members `setWidth` and `setHeight` are referred to together twice by different program elements (`resetConfig` and `correctAspectRatio`). This means they are closely related. In this example, only these elements are together in the same cluster likely due to the weight of the edge.

The idea of applying clustering to this representation is to identify close feature-interface members as clusters, such that the overall number of program elements that must be considered during feature maintenance will be reduced guided by the clusters of the segregated feature interfaces. Figure 5 (right part) illustrates a possible clustering the MRG extracted from the feature interface SCREEN (cf. Section 2.3).

4. Methodology

In this section, we describe our study in terms of its goal (Section 4.1), the subject system used to evaluate the segregation of feature interfaces (Section 4.2), and the evaluation procedure used to conduct the study (Section 4.3).

4.1 Research Question

The goal of our study is to analyse how well segregated feature interfaces can be used to perform maintenance tasks against the original feature interface. We use the co-changes of interface members in commits as our oracle for the relevant subset of interface members. Then, we compare the number of interface members likely to be unnecessarily considered by developers to the clusters of the segregated feature interface, on the one hand, and to the original interface, on the other hand. Therefore, we formulate the following research question:

RQ: *How well can segregated feature interfaces be used to perform code maintenance in software product lines, compared to the original feature interfaces?*

4.2 Subject System

We selected *BusyBox*[2] as a paradigmatic case study, representing many other product-line implementations based on conditional compilation [19]. BusyBox is a real-world resource-efficient product line of UNIX utilities implemented in C. BusyBox runs in a variety of POSIX environments, such as Linux, Android, FreeBSD, and others [19]. Variability in BusyBox includes both variability at the composition level, automated by the build system, and variability at source-code level, encoded with preprocessor directives. In our study, we focus on variability at the source-code level. We selected 10 major versions of BusyBox. Table 1 provides general data about the 10 versions, including lines of code (KLOC), number of features (# Features), and number of feature dependencies (# Dependencies).

[2] `http://busybox.net/`

Table 1. BusyBox releases selected for our study.

Release	KLOC	# Features	# Dependencies
1.13	183	646	630
1.14	188	668	620
1.15	185	696	671
1.16	191	722	702
1.17	196	738	681
1.18	209	759	718
1.18.5	199	759	719
1.19	192	776	761
1.20	194	781	762
1.21	195	766	749
mean	**193**	**731**	**701**

We chose BusyBox because of three main reasons: (i) BusyBox is a widely-known system already used in different studies, and its sources (and commits) are openly accessible; (ii) it is a real-world product line with a large number of features and feature dependencies; and (iii) BusyBox has been developed by an open-source community since 1999 and is still evolving. Furthermore, we aim at conducting a longitudinal study of one single product line. The idea was to carefully analyse the feature interfaces proposed before and after clustering to discuss in-depth the implications of our results and answer the research question. To do so, we needed to understand the semantics of the interface members as well as the semantics of feature dependencies. This would not be possible if we conduct a wider and more superficial study using many different product lines.

4.3 Evaluation Procedure

Our study is divided in three major phases: (i) data extraction of feature dependencies, feature interfaces, member relationship graphs (MRG), and co-changes of interface members, (ii) clustering, and (iii) evaluation. In the following, we give more details on the three major phases of our study.

4.3.1 Data Extraction

We extracted 10 releases of BusyBox based on the corresponding tags in the version-control system. We scanned each C file in the selected releases (in total, 6,858 source-code files) to mine feature dependencies in the source code using the tool TypeChef [20]. It is important to mention that we have a feature model for BusyBox, which we used by supplying it to TypeChef. Therefore, we only consider configuration-relevant features in our approach.

Feature-dependency and feature-interface extraction. To extract feature dependencies, we analyse the variability-aware control-flow graph [23] of each release of BusyBox using TypeChef. Each node of the control-flow graph, representing a program element, is associated with a feature or set of features. Feature dependencies are identified by control flows between nodes (i.e., program elements) of different features. In the context of our study, we consider a dependency between features A and B if: (i) A references an attribute of B, or (ii) A calls a method of B. Once we extracted all nodes participating in feature dependencies, we group them in dependee and dependent features. Then, we classify nodes of a feature that have incident edges (i.e., they are used by another feature) as members of such feature's provided interface

Feature-interface filtering and MRG construction. Once we extracted all feature interfaces, we select only interfaces with more than one member, as we do not need to segregate the feature interface for these cases. After that, we construct the MRG (cf. Section 3.2). To this end, we developed a TypeChef extension that takes the variability-aware CFG output from TypeChef and constructs the MRG from it. Furthermore, we used R scripts[3] for graph processing. The aim of this step is to relate feature interface members. Edges between nodes in the MRG, therefore, represent the relationships that exist between the members of a feature interface. In this case, the topology of the graph becomes informative regarding the structure of feature dependencies.

Co-change extraction. The last step of data extraction is to mine changes on interface members from the BusyBox repository using Codeface[4] [15]. The idea is to identify interface members that co-change (i.e., simultaneously change) in a single commit.

Overall, we extracted 2,286 feature interfaces out of 7,311 features. After filtering out feature interfaces with only one member, the resulting number of feature interfaces was 650. The total number of feature-interface members under analysis was 3,154; the biggest feature interface contains 46 members. The total number of analysed feature dependencies is 7,013, while the maximum number in a release is 762. Regarding the co-changes extracted from the BusyBox repository, we extracted a total of 3,382 changes in program elements, comprising 2,592 commits for the 10 releases of BusyBox.

4.3.2 Clustering

We use the Markov Cluster (MCL) algorithm [12] to cluster the extracted MRGs. MCL is a cluster algorithm for graphs based on the simulation of stochastic flows in graphs. It is based on the graph-clustering paradigm, which postulates that natural groups in graphs have the following property: *"A random walk in a graph that visits a dense cluster will likely not leave the cluster until many of its vertices have been visited"* [12]. Natural groups (clusters) in a graph are characterised by the presence of many edges (or more weighted edges) between the members of that cluster. In particular, this number should be high, relative to node pairs that span different clusters. In other words, random walks on the graph will infrequently go from one cluster to another.

The MCL algorithm finds cluster structures in graphs by a mathematical bootstrapping procedure. The MCL algorithm respects edge weights and considers them as a means of similarity. The process deterministically computes (the probabilities of) random walks through edges and edge weights in the graph. This way, the algorithm uses stochastic matrices (also called Markov matrices) that capture the mathematical concept of random walks on a graph.

The MCL algorithm simulates random walks in a graph by alternation of two operators called expansion and inflation. *Expansion* computes random walks of high length, which means random walks with many steps. An expansion associates new probabilities to all pairs of nodes, where one node is the point of departure and the other is the destination. Since high-length paths are more common inside clusters than between different clusters, the probabilities associated with node pairs lying in the same cluster will be relatively large [12]. *Inflation* will have the effect of boosting the probabilities of intra-cluster walks and will relegate inter-cluster walks. Iterating expansion and inflation results in the separation of the graph into different segments. The collection of resulting segments is then interpreted as a clustering [12].

We chose the MCL cluster algorithm because of several advantages. First, the algorithm fits our idea of a cluster in the MRG: functions that are densely linked and, hence, are used together in a workflow are in the same cluster. A developer will likely use many other functions of a single cluster in a workflow before using a function of a different cluster, thus matching the MCL algorithm property. Second, it performs two simple algebraic operations on matrices. There are no high-level procedural instructions for assembling, joining, or splitting of groups. In other words, the MCL algorithm was designed with the consideration of scalability, which is very important for our use case. Third, in the MCL algorithm, the number of clusters can not be specified in advance, differently from other well-known clustering algorithms (e.g., K-means [25], Lloyd's [24], etc.).

The MCL algorithm was applied to the MRGs of the BusyBox using an R library called MCL[5]. To calibrate the MCL algorithm regarding its only parameter (i.e., the inflation parameter), we used a tool called *clm dist*, implemented by the creator of the MCL algorithm and located in the same library of the algorithm. The tool can suggest the value of the parameter based on the number of elements to be clustered. After generating the clusters for all BusyBox releases, we executed the clm dist tool. The value 2 was the suggested value to be used in our case.

4.3.3 Evaluation

To validate the segregation of feature interfaces produced by the MCL algorithm, we used the *Jaccard distance* between the feature interfaces (segregated and original), on the one side, and co-changes of interface members within a commit, on the other side. The Jaccard distance is a statistic used for measuring the dissimilarity between finite sample sets, and it is defined by subtracting the size of the intersection from

[3] `http://www.r-project.org/`

[4] `http://siemens.github.io/codeface/`

[5] `https://cran.r-project.org/web/packages/MCL/`

the size of the union, divided by the size of the union of the sample sets:

$$d_J(A,B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|} \quad (1)$$

Values of the Jaccard distance are bound between 0 and 1. Values closer to 1 mean a higher dissimilarity between the sets. Values closer to 0 mean lower dissimilarity between the sets. By using the Jaccard distance, we are able to find the number of interface members unnecessarily considered by the developer during a single maintenance task based on our ideal model of clusters (i.e., the commits of the product line). For instance, in our example presented in Section 2.3, two interface members would change in a commit (our ideal cluster). In the original interface, the Jaccard distance would be $d_J(A,B) \approx 0.67$, being A the set of members of a commit and B the set of members of a specific cluster. In the original interface, we consider the whole interface as a single cluster. Therefore, we are able to use the results and make statements such as "67% of the members of a feature interface (or a cluster of a feature interface) have been unnecessarily considered by developers during a maintenance task".

Our goal is to reduce the number of interface members that are likely to be unnecessarily considered by developers during maintenance. In our example presented in Section 2.3, the members `setBrightness`, `getFrequency`, `getResolution` and `setContrast` would be unnecessarily considered by a developer in the context of this maintenance task. Those interface members increase the complexity of reasoning about feature and their dependencies. Therefore, the more dissimilar a cluster of interface members is, when compared to the commit changes, the worse is the feature-interface organisation (segregated or original interface).

We conducted the comparison of feature interfaces with the co-changes as follows. For each set of co-changes of interface members, we compute the Jaccard distance with every cluster of feature interfaces generated by the MCL algorithm. In an original feature interface, we consider the whole interface as a single cluster to simulate the mindset of members to be considered by the developer in a maintenance task. We calculate the Jaccard distance between all elements of the commit belonging to a single feature and each cluster of the feature interface (original and segregated). To have a single value for each interface, the average Jaccard distance is calculated using the values found for each interface cluster. Finally, the Jaccard distance found for each commit is used to calculate the average of a release. Therefore, we have a Jaccard distance for each release and the type of feature interface (original and segregated).

5. Results

The goal of our study is to analyse the amount of code a developer needs to consider in a change task with a segregated interface compared to the original interface. A high Jaccard distance indicates the percentage of interface members that might have been inspected unnecessarily during product-line maintenance based on commit changes. Table 2 shows the results of the Jaccard distance for both types of feature interfaces for each release analysed of BusyBox.

Looking at Table 2, we notice that the values for segregated interfaces ranges between 0.1 and 0.4. That is, in the worst case, the number of feature members likely to be unnecessarily considered by developers is, at most, 40% of the interface members. The concentration of Jaccard distance distribution for the original interface close to 0.9 means that approximately 90% of the feature-interface members were not related to the commits.

The Jaccard distance for segregated interfaces is always lower when compared to the original interfaces. In addition, the difference of mean between segregated and original interface is almost 62%. This means that, using segregated interfaces, reduces the number of interface members unnecessarily analysed by 62%. So, since we are using the Jaccard distance in our study as a measure of the number of elements likely to be unnecessarily revisited during maintenance, we can say the number of feature interface members to be considered in a maintenance task is lower than the one in a original interface.

To gain confidence in this result, we computed the difference between Jaccard distances for segregated and original interfaces using the paired Mann-Whitney test [28] and the paired Cliff's Delta effect size [14]. We used these tests because the data are not normally distributed. First, we used the Mann-Whitney test to analyse whether there is a significant difference between the Jaccard distances of clustered and non-clustered interfaces. Significant differences are indicated by *p-values* lower than 0.01. Then, we use the Cliff's Delta effect size (*d*) to measure the magnitude of the difference between the Jaccard distance between segregated and original interfaces. Cliff's Delta is bound between +1 and -1. Values close to +1 mean that all selected values from one group are higher than the selected values in the other group, values close to -1 when the reverse is true. The value 0 expresses two overlapping distributions. The effect size is considered negligible for $|d| < 0.147$, small for $0.147 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.47$, and large for $|d| \geq 0.47$ [14].

The *p-value* of the Mann-Whitney test is smaller than 9e-5. This result suggests that the difference between the

Table 2. Jaccard distances for both types of interfaces.

Release	Segregated Interface	Original Interface
1.13	0.2263464	0.9015086
1.14	0.3745138	0.8869625
1.15	0.2767011	0.8712999
1.16	0.1973850	0.8545383
1.17	0.2588888	0.9149021
1.18	0.2375843	0.9578755
1.18.5	0.3611111	0.9007790
1.19	0.1918869	0.8614765
1.20	0.4325397	0.8878205
1.21	0.1071429	0.8289116
mean	**0.2764**	**0.8866**

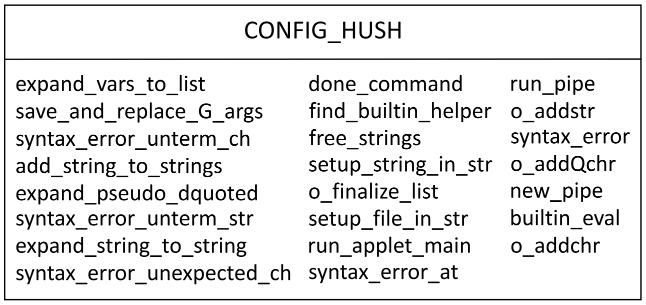

Figure 6. Original interface of CONFIG_HUSH.

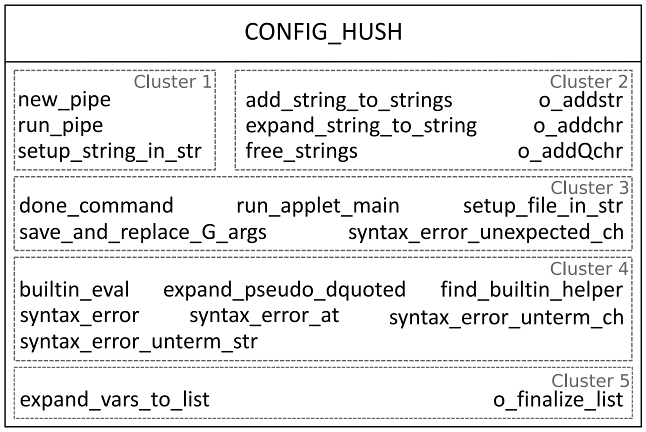

Figure 7. Segregated interface of CONFIG_HUSH.

two distributions is statistically significant. The Cliff's Delta measured on both distributions is $d = -1$. This means that the effect size is statistically large and the difference between both distributions is relevant. In other words, based on the results, we argue that segregated feature interfaces have the potential to support developers in reasoning about features and their dependencies during maintenance of product lines.

To answer the research question, the results of the Jaccard distance as well as the p-value and Cliff's Delta effect size allow us to say the segregated feature interfaces can better support developers to perform maintenance tasks in software product lines when compared against original feature interfaces.

6. Discussion

6.1 The Impact of Segregating Feature Interfaces

The difference when comparing the Jaccard distance between segregated and original interfaces (Table 2) indicates a significant reduction of the number of members that need to be considered when developers need to reason about product-line maintenance ($\approx 62\%$ of reduction, on average).

As an example to illustrate such reduction, commit 2127 from release 1.17 of BusyBox touched only one interface member (the method `expand_vars_to_list`) of feature CONFIG_HUSH. This feature has 23 interface members, in total, though. Figure 6 illustrates the original interface of the feature CONFIG_HUSH.

In a original interface, a simple change in `expand_vars_to_list` would demand that the other 22 (out of 23) interface members need to be considered by the developer when reasoning about the effects of this change—that is, about 96% of the members would have been unnecessarily considered in commit 2127. In the segregated interface of feature CONFIG_HUSH, there are five clusters. The cluster containing the member touched in commit 2127 (`expand_vars_to_list`) has only two members. Figure 7 illustrates the segregated feature interface of feature CONFIG_HUSH. With our segregated interface, among the 23 members, the developer needs to consider only one member more (`o_finalize_list`) out of 23 interface members (i.e. $\approx 4\%$). Members of other clusters are unlikely to be related to `expand_vars_to_list`, thus do not need to be considered by developers.

Despite the benefits, our approach may produce wrong results. Figure 7 illustrates one possible improvement in our clustering approach. The member `syntax_error_unexpected_ch` seems to be located in the wrong cluster. Members starting with the prefix `syntax_error_` may be related and should be considered together by developers. However, four members starting with the prefix `syntax_error_` are located in one cluster while `syntax_error_unexpected_ch` is in a different one. A possible reason for the wrong location is related to the MRG (cf. Section 3.2). The MRG represents relationships between interface members found in the source code. In this specific case, the weight of the edge between the members starting with the prefix `syntax_error_` and `syntax_error_unexpected_ch` had a low value. As a consequence, the clustering algorithm segregated parts of the MRG, thus relating `syntax_error_unexpected_ch` to the other members. As future work, we shall consider the similarity of names as a variable in our MRG (i.e., semantic coupling). Similar names could have different weights in the MRG, thus avoiding segregation of related members.

6.2 Stability of Interfaces

An important issue for the adoption of our approach is the stability of the segregated interfaces. A stable interface is an interface that is not subject to significant changes of its members during evolution. Frequent changes of clustered interfaces during evolution could be a barrier for the adoption of our solution. Developers would need to become familiar with a new organisation of interfaces again and again. However, we observed that, since there are no considerable changes in the structure of the product line, the clustered interfaces also does not suffer many changes in their structure. This fact can be observed in Table 3. We can notice that situation by observing that the number of features, number of feature members, and the average number of members in a cluster do not vary widely during the evolution of BusyBox.

Table 3. Information about the number of feature interface members of BusyBox.

Release	# of features	# of feature members	Avg. # of feature members per cluster
1.13	54	263	3.81
1.14	57	309	3.96
1.15	59	315	3.89
1.16	60	314	3.83
1.17	55	304	4.11
1.18	58	325	4.11
1.18.5	59	326	4.07
1.19	62	334	4.07
1.20	61	333	4.06
1.21	61	331	4.04
mean	**58.6**	**315.4**	**3.9**

For instance, Table 3 shows a variation of 7 features from the first release (54 features) to the last release (61 features). In addition, the number of feature members (variation of 68 members) remained stable. It is also important to mention that we observed the members themselves also experienced only few changes along the evolution. So, we can consider that the evolution of BusyBox was not subject to considerable changes. In this case, we should expect that the average number of members in a cluster remains stable. In fact, the average number of members in a cluster showed a very low variation along the evolution of BusyBox. Moreover, the structure of each cluster also did not suffer considerable changes along the evolution. Based on this result, we conclude that it is likely that features and their dependencies stabilise as the product-line evolution also stabilise. As a consequence, it is also likely that segregated interfaces also stabilise with time. In this context, we argue that the adoption of our approach in evolving product lines does not demand a lot of effort from developers to become familiar with segregated interfaces.

6.3 Nature of Changes

Another important factor that impacts our results are the commits used to validate our approach. Depending on characteristics of the commit, the Jaccard distance for some segregated interfaces tends to be high when compared to the majority of the results. For instance, commit 2121 involved changes to 20 interface members of 9 different features. One can notice that many interface members (possibly from multiple features) have been changed in this single commit. In this commit, the number of features being changed is exceptionally high. The reasons for a coarse-grained commit like that are many. For example, intrinsic characteristics of the maintenance task may demand changes in several interface members. The problem is that, the more members are changed in a single commit, the lower is the chance that a cluster comprises all these members. Developers might have to analyse several interface clusters within a feature to comprise all members that must be considered. Even worse, when commits involve changes of members from different features, it is challenging to reach a good result, since we are using commits as our ideal model of cluster. In the example presented here, we can notice that the number of members changed in a single commit (20 interface members of 9 different features) is higher than the average number of members within a cluster (3.9 members per cluster, see Table 3). As a consequence, the Jaccard distance measured for this commit is expected to be high. Therefore, we can say that coarse-grained commits can compromise the effectiveness of our solution.

Table 4. Information about the number of program elements changed per commit.

Release	# of commits	# of elements changed	Avg. # of elements changed per commit
1.13	384	626	1.63
1.14	272	535	1.97
1.15	438	775	1.77
1.16	422	646	1.53
1.17	356	485	1.36
1.18	5	30	6.00
1.18.5	49	48	0.98
1.19	229	283	1.24
1.20	166	197	1.19
1.21	271	257	0.95
mean	**259.2**	**388.2**	**1.86**

Despite of the aforementioned issues, the majority of the commits usually comprise changes in program elements that are somehow related. Table 4 provides information regarding the number of commits, number of program elements changed in a release, and average of program elements changed per commit. One can observe that, in general, releases with an average number of program elements per commit greater than the total average number (i.e., 1.86 elements per commit) had the longest Jaccard distances (see Table 2). In this case, we can argue that only exceptional cases of commits may compromise our approach. Still, the results are better than the obtained results for original interfaces.

7. Threats to Validity

Conclusion validity. Conclusion validity concerns the relationship between the treatment and the outcome. In this study, potential threats arise from the statistical tests used to support our conclusions. To mitigate this threat wherever possible, we used statistical tests obeying the characteristics of our data. In particular, we used non-parametric tests, which do not make any assumption on the underlying data distribution regarding variances and types.

Internal validity. Internal validity is the degree to which conclusions can be drawn about the causal effect of independent variables on the dependent variables. In our study, potential threats arise from the tuning of the MCL algorithm that may affect the results. Changing the only parameter of the algorithm (i.e., the inflation parameter) will affect the granularity of the clusters, thus affecting the Jaccard distances used to compare the results. We mitigated this threat by calibrating the algorithm to use the recommended value indicated by the author of the algorithm, which is based on the number of members to be clustered (Section 4.3.2). A further threat arises from the fact that we focus on conditional compilation as the variability mechanism for implementing features in the source code. With conditional compilation, features are often tangled and scattered in the source code. This choice means that a feature may have (i) many members in its interface, and (ii) many of them may not be cohesively related to each other and, therefore, may not be relevant for each change propagation. This situation could have been different if we had decided to analyse product lines implemented using a compositional approach (e.g., aspect-oriented programming). However, we argue that conditional compilation is the most-widely used mechanism to implement product-line features [16]. Moreover, BusyBox (Section 4.2) is an industrial project. So, we believe the results extracted from this product line can be a first step towards the generalisation of the results. In fact, BusyBox contains categories of features implemented in a wide range of different ways: from fully-modularised features to scattered and tangled features.

External validity. Threats associated with external validity concern the degree to which the findings can be generalised to the wider classes of subjects from which the experimental work has drawn a sample [31]. In our work, this is a particularly important threat to validity in face of the wide range of diverse product lines implemented using conditional compilation. In the experiment reported here, this threat to validity is somewhat mitigated by the fact that we selected BusyBox (Section 4.2) to conduct our study. BusyBox can be considered as a paradigmatic case study, which represents many other product-line implementations based on conditional compilation due to, for example, its size, number of features, and number of valid configurations [19]. In addition, since we are analysing different releases of the same product line, there is no risk that the variation due to individual differences of releases is larger than due to the treatment.

8. Related Work

Feature modularity. Feature modularity has been a long-standing goal of feature-oriented software development [4]. While some researchers view features as modular units of behavior and composition, others pointed out that, at the source-code level, most implementation mechanisms provide merely syntactic compositions, and thus lack proper interface abstractions and modular reasoning. In this context, Kästner et al. pinpoint two different notions of feature modularity: one based on locality and cohesion, another based on information hiding and interfaces [18]. Modularity means locality and cohesion when a feature is viewed as a unit of composition that has the goal of making itself explicit in design and implementation [4]. Therefore, everything related to a feature is placed into a separate structure called feature module [3].

Another view of feature modularity is rooted in the concept of information hiding and interfaces. The idea is to distinguish between an internal and an external part of a feature module. The internal part is hidden. The external part is called interface and controls the communication between different feature modules [18]. Most of the work on feature modularity has focused on locality and cohesion of features as a criterion for system decomposition and assembly. Examples of approaches for improving feature modularity include architecture-based product lines (based on frameworks or components) [8], feature-oriented programming [32], aspectual feature modules [3], and superimposition [5]. Despite the improvement of feature modularity in those cases, simple solutions such as conditional compilation prevail in practice [17]. Nevertheless, there is a lack of studies driving efforts towards feature interfaces in solutions based on (i) locality and cohesion, and (ii) in widespread adopted solutions such as conditional compilation. Our work suggests a way of organising feature interface. This organisation is based on structural properties of feature dependencies agnostic of the implementation approach. This way, our work enhances feature modularity by providing organised feature interfaces to support change propagation in both compositional and annotative approaches.

Finally, there is substantial progress in solving problems that threat modularity of features. The feature-interaction problem is considered a major threat to modularity in that the behavior of one feature may be affected by the presence of another feature [6]. So, developers must analyse the consequences of all possible feature interactions to find the undesired ones. In other words, the feature-interaction problem also hinders independent feature maintenance. Some studies address with feature interactions and its problems [2, 6, 9]. Despite the similar focus of these studies to our approach, none of them aims at improving feature interfaces to support product-line maintenance.

Feature interfaces. Ribeiro et al. proposed an approach for generating interfaces of product-line features [33, 34]. They defined the concept of *emergent interfaces* for product lines implemented with conditional compilation. Their approach aims at establishing interfaces between features on demand – based on source code –, with the goal of preventing developers from breaking other features when performing maintenance tasks. Despite the generation of interfaces, this approach generates only interfaces related to specific parts of the source code that are of interest, and thus do not allow having a global view of the system. In other words, unlike our work, emergent interfaces do not address the problem of large/monolithic interfaces by segregating them.

Schröter et al. proposed the idea of *feature-context interfaces* for composition-based product lines [36]. The idea is to show the developer of a feature the interface members of other features that can be safely accessed from the current context without risking dangling references. The reasoning about the presence of the feature interface members is based on the feature model. We also use the current context in our approach, but, in contrast, we aim at identifying related interface members within the same feature interface. Furthermore, we focus on preprocessor-based product lines and we do not use the feature model to reason about the presence of other features in the current context.

There is further work that concentrates on interfaces in approaches used to implement product lines or in variability-aware analysis approaches supported by interfaces. For example, Kästner and colleagues propose a variability-aware module system for product lines [19]. This approach infers interfaces for modules focusing on type checking of product-line configurations. Kiczales and Mezini propose aspect-aware interfaces, computing an aspect's dependencies on a system's join points and displaying these dependencies as annotations on the explicit interfaces of advised code [21]. Li et al. propose a new methodology to verify cross-cutting features as open systems by using a model of semantic interfaces that supports automated, compositional, and feature-oriented model checking [22]. However, none of these studies considers interfaces in the light of for supporting maintenance tasks.

Automatic Modularisation. The problem of automatic modularisation (also referred to as automatic-system clustering) has been extensively studied [27]. The idea is to help developers creating a good mental model of system's organisation. The field was established by the seminal work of Mancoridis et al. [26, 27, 35]. In this work, the authors use a hill-climbing algorithm as the primary search technique for automated software-module clustering. Several other meta-heuristic search techniques have been applied, including genetic algorithms [26]. However, all these studies focus on the most common application of clustering in automatic modularisation: software-module clustering. We apply the idea of clustering to enhance the modularity of features by segregating feature interfaces. Regarding the automatic modularisation of interfaces, after the introduction of the interface segregation principle [29], some studies have proposed ways of segregating feature interfaces. For instance, in a recent work, Romano et al. propose a way of refactoring fat interfaces (i.e., interfaces whose clients invoke different subsets of their members) of classes of object-oriented programs using genetic algorithms [35]. We do not consider refactoring ill-defined interfaces. Furthermore, identifying the interfaces of classes in an object-oriented program is much easier than in conditional-compilation-based product lines since all the classes are already modularised. We focus on segregating interfaces of product-line features instead of classes' interfaces. Our study was the first to systematically investigate and compare the co-relation of segregated and original interfaces.

9. Conclusion

In practice, the lack of feature modularity complicates the maintenance of complex preprocessor-based product lines [4]. As features can be heavily scattered across the source code, developers need to consider many program elements to understand the feature itself and its dependencies. Developers usually cope with the complexity during maintenance using interfaces. Analogously, a feature interface contains all program elements of the source code belonging to the feature that provides external access for other features. As reasoning about all feature interface members alone is complex, we argue that often only a subset of interface members is relevant to a single maintenance task.

In this paper, we proposed a technique for automating the segregation of feature interfaces to maximise the cohesion of interface members. We proposed a representation (the member relationship graph, MRG) that is able to capture the collaboration between interface members and apply a clustering algorithm on it to group highly-related members. To evaluate our approach, we selected ten versions of Busybox, we constructed the MRGs for all versions and reorganised the feature interfaces using our technique. Using changes in the commits of the subject system as a foundation, we analysed how well segregated feature interfaces can support developers with maintenance tasks in comparison to the original feature interfaces. In other words, we analysed the number of interface members likely to be unnecessarily considered by developers in both types of feature interfaces. The results of our study show a pronounced difference of approximately 62% in favour of segregated interfaces regarding the reduction of interface members likely unnecessarily considered by developers during maintenance. Furthermore, we observed that our segregated interfaces stabilise with time. So, the adoption of our approach in product-line evolution does not demand a lot of effort from developers to become familiar with our segregated interface after every change in a release. Finally, we learned that segregating feature interfaces may help developers to refer to members that potentially correspond to a maintenance task. This way, the effort to reason modularly about features can be reduced, and thus the maintenance-task difficulty and number of potentially related program elements to observe.

As future work, we plan (i) to analyse other software product lines with our approach, (ii) to use product lines implemented with other approaches (e.g., aspect-oriented programming, feature-oriented programming, etc.), (iii) to use other maintenance factors (e.g., bugs) instead of co-changes to assess the usefulness of segregated interfaces, (iv) to evaluate our approach using different clustering algorithms and other techniques to improve the clustering based on other properties (e.g., similarity of methods' names [1]), and (v) to incorporate approaches to select only relevant commits to be considered in our study (e.g., [38]).

10. Acknowledgements

We thank T. Bock and M. Joblin for their fruitful comments. This research was supported by CAPES (14565-13-7 and project numbers 483882/2009-7, 485348/2011-0, 175956), CNPq (305526/2009-0, 308490/2012-6, 141308/2015-0), FAPERJ (E-26/102.211/2009, E-26/010.001421/2015), PUC-Rio (productivity grant), and the German Research Foundation (AP 206/2, AP 206/4 and AP 206/5).

References

[1] N. Alhindawi, N. Dragan, M. Collard, and J. Maletic. Improving Feature Location by Enhancing Source Code with Stereotypes. In *Proc. ICSM*, pages 300–309. IEEE, 2013.

[2] S. Apel and D. Beyer. Feature Cohesion in Software Product Lines: An Exploratory Study. In *Proc. ICSE*, pages 421–430. ACM, 2011.

[3] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An Algebra for Features and Feature Composition. In *Proc. AMAST*, pages 36–50. Springer, 2008.

[4] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013.

[5] S. Apel, C. Kästner, and C. Lengauer. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *TSE*, 39(1):63–79, 2013.

[6] S. Apel, A. von Rhein, T. Thüm, and C. Kästner. Feature-Interaction Detection Based on Feature-Based Specifications. *Computer Networks*, 57(12):2399 – 2409, 2013.

[7] S. Apel, A. von Rhein, P. Wendler, A. Grösslinger, and D. Beyer. Strategies for Product-Line Verification: Case Studies and Experiments. In *Proc. ICSE*, pages 482–491. IEEE, 2013.

[8] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2 edition, 2003.

[9] D. Batory, P. Höfner, and J. Kim. Feature Interactions, Products, and Composition. In *Proc. GPCE*, pages 13–22. ACM, 2011.

[10] B. B. P. Cafeo, E. Cirilo, A. Garcia, F. Dantas, and J. Lee. Feature Dependencies as Change Propagators: An Exploratory Study of Software Product Lines. *Information and Software Technology*, 69(1):37–49, 2016, online first.

[11] P. C. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, 2001.

[12] S. Dongen. A Cluster Algorithm for Graphs. Technical Report CAG-868986, National Research Institute for Mathematics and Computer Science, 2000.

[13] B. S. Everitt, S. Landau, and M. Leese. *Cluster Analysis*. Wiley Publishing, 4th edition, 2009.

[14] R. J. Grissom and J. J. Kim. *Effect Sizes for Research: A Broad Practical Approach*. Lawrence Erlbaum, 2nd edition, 2005.

[15] M. Joblin, W. Mauerer, S. Apel, J. Siegmund, and D. Riehle. From Developer Networks to Verified Communities: A Fine-Grained Approach. In *Proc. ICSE*, pages 563–573. IEEE, 2015.

[16] C. Kästner and S. Apel. Virtual Separation of Concerns – A Second Chance for Preprocessors. *Journal of Object Technology*, 8(6):59–78, 2009.

[17] C. Kästner, S. Trujillo, and S. Apel. Visualizing Software Product Line Variabilities in Source Code. In *Proc. ViSPLE*, pages 303–312, 2008.

[18] C. Kästner, S. Apel, and K. Ostermann. The Road to Feature Modularity? In *Proc. SPLC*, pages 51–58. ACM, 2011.

[19] C. Kästner, K. Ostermann, and S. Erdweg. A Variability-Aware Module System. In *Proc. OOPSLA*, pages 773–792. ACM, 2012.

[20] A. Kenner, C. Kästner, S. Haase, and T. Leich. TypeChef: Toward Type Checking #Ifdef Variability in C. In *Proc. FOSD*, pages 25–32. ACM, 2010.

[21] G. Kiczales and M. Mezini. Aspect-oriented Programming and Modular Reasoning. In *Proc. ICSE*, pages 49–58. ACM, 2005.

[22] H. C. Li, S. Krishnamurthi, and K. Fisler. Interfaces for Modular Feature Verification. In *Proc. ASE*, pages 195–204. IEEE, 2002.

[23] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable Analysis of Variable Software. In *Proc. ESEC/FSE*, pages 81–91. ACM, 2013.

[24] S. Lloyd. Least Squares Quantization in PCM. *TIT*, 28:129–137, 2006.

[25] J. Macqueen. Some methods for classification and analysis of multivariate observations. In *5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[26] K. Mahdavi, M. Harman, and R. M. Hierons. A Multiple Hill Climbing Approach to Software Module Clustering. In *Proc. ICSM*, pages 315–324. IEEE, 2003.

[27] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner. Using Automatic Clustering to Produce High-Level System Organizations of Source Code. In *Proc. IWPC*, pages 45–52, 1998.

[28] H. B. Mann and D. R. Whitney. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947.

[29] R. C. Martin. The Interface Segregation Principle: One of the many Principles of OOD. *C++ Report*, 8:30–36, 1996.

[30] D. L. Parnas, P. C. Clements, and D. M. Weiss. The Modular Structure of Complex Systems. In *Proc. ICSE*, pages 408–417. IEEE, 1984.

[31] K. Praditwong, M. Harman, and X. Yao. Software Module Clustering as a Multi-Objective Search Problem. *TSE*, 37(2): 264–282, 2011.

[32] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. ECOOP*, pages 419–443. Springer, 1997.

[33] M. Ribeiro, F. Queiroz, P. Borba, T. Tolêdo, C. Brabrand, and S. Soares. On the Impact of Feature Dependencies when Maintaining Preprocessor-Based Software Product Lines. In *Proc. GPCE*, pages 23–32. ACM, 2011.

[34] M. Ribeiro, P. Borba, and C. Kästner. Feature Maintenance with Emergent Interfaces. In *Proc. ICSE*, pages 989–1000. ACM, 2014.

[35] D. Romano, S. Raemaekers, and M. Pinzger. Refactoring Fat Interfaces Using a Genetic Algorithm. In *Proc. ICSME*, pages 351–360. IEEE, 2014.

[36] R. Schröter, N. Siegmund, T. Thüm, and G. Saake. Feature-Context Interfaces: Tailored Programming Interfaces for Software Product Lines. In *Proc. SPLC*, pages 102–111. ACM, 2014.

[37] S. Trujillo, D. Batory, and O. Diaz. Feature Refactoring a Multi-Representation Program into a Product Line. In *Proc. GPCE*, pages 191–200. ACM, 2006.

[38] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining Version Histories to Guide Software Changes. In *Proc. ICSE*, pages 563–572. IEEE, 2004.