View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Open Repository and Bibliography - Luxembourg

Feature location benchmark for extractive software product line adoption research using realistic and synthetic Eclipse variants

Jabier Martinez[a,b,*], Tewfik Ziadi[b], Mike Papadakis[a], Tegawendé F. Bissyandé[a], Jacques Klein[a], Yves le Traon[a]

[a]*SnT, University of Luxembourg, Luxembourg*
[b]*LiP6, Sorbonne Universités, UPMC University Paris 6, France*

Abstract

It is common belief that high impact research in software reuse requires assessment in non-trivial, comparable, and reproducible settings. However, software artefacts and common representations are usually unavailable. Also, establishing a representative ground truth is a challenging and debatable subject. Feature location in the context of software families is a research field that is becoming more mature with a high proliferation of techniques. We present EFLBench, a benchmark and a framework to provide a common ground for this field. EFLBench leverages the efforts made by the Eclipse Community which provides feature-based family artefacts and their implementations. Eclipse is an active and non-trivial project and thus, it establishes an unbiased ground truth which is realistic and challenging. EFLBench is publicly available and supports all tasks for feature location techniques integration, benchmark construction and benchmark usage. We demonstrate its usage and its simplicity and reproducibility by comparing four techniques in Eclipse releases. As an extension of our previously published work, we also contribute an approach to automatically generate Eclipse variants to benchmark feature location techniques in tailored settings. We present and discuss three strategies for this automatic generation and we present the results using different settings.

Keywords: Feature location, software families, Eclipse, benchmark, software product lines, static analysis, information retrieval

1. Introduction

Feature location focuses on mapping features to their concrete implementation elements in the software artefacts. This activity is important during software maintenance for determining relevant elements for a modification task [1,

*Corresponding author: jabimail@gmail.com

2]. In the context of this paper, instead of a single artefact, we consider a family of artefact variants where feature location is an essential activity of extractive processes towards systematic reuse [3], notably in leveraging a set of legacy variants for the adoption of a Software Product Line (SPL) [4, 5, 6]. An SPL is formally defined as *"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, and that are developed from a common set of core assets in a prescribed way"* [4]. Feature location is needed to identify the implementation elements that will be used to create the reusable assets of each feature. Given the increasing interest by the research community on this subject [7, 8], feature location benchmarks are required to enable an intensive experimentation of the techniques. This paper is an extension paper of our benchmark framework [9] which elaborate further on the need to empirically evaluate and compare the strengths and weakness of the techniques in different scenarios. However, **comparing and experimenting with feature location techniques is challenging** because of the following reasons:

- *Most of the research prototypes are either unavailable or hard to configure.* There exists a lack of accessibility to the tools implementing each technique with its variants abstraction and feature location phases.
- *Most of the tools are strongly dependent on specific artefact types* that they were designed for (e.g., a given type of model or programming language).
- *Performance comparison requires common settings and environments.* There exist difficulties to reproduce the experimental settings to compare performance.

Given that common case study subjects and frameworks are in need to foster the research activity [10], we identified two requirements for such frameworks in feature location:

- *A standard case study subject:* Subjects that are non-trivial and easy to use are needed. This includes: 1) A list of existing features, 2) for each feature, a group of elements implementing it and 3) a set of product variants accompanied by the information of the included features.
- *A benchmarking framework:* In addition to the standard subjects, a full implementation allowing a common, quick and intensive evaluation is needed. This includes: 1) An available implementation with a common abstraction for the product variants to be considered by the case studies, 2) easy and extensible mechanisms to integrate feature location techniques to support the experimentation, and 3) predefined evaluation metrics to draw comparable results.

The contributions of this paper are:

- We present the **Eclipse Feature Location Benchmark (EFLBench)** and examples of its usage. We propose a standard case study for feature location and a benchmark framework using Eclipse variants, their features and their associated plugins. We implemented EFLBench within the Bottom-Up Technologies for Reuse framework (BUT4Reuse) which allows a quick integration of feature location techniques [11]. By integrating a feature location technique in this generic and extensible framework, the technique could be applied in other artefact types beyond the experimentation with Eclipse variants within EFLBench.
- We present the **automatic generation of Eclipse variants** as part of EFLBench capabilities to construct tailored benchmarks. This enables the evaluation of techniques in different scenarios to show their strengths and weaknesses. This is the significant increment from our previous work [9]. The new contribution extends the use of the benchmark beyond the official Eclipse releases providing three strategies to tailor the settings of the benchmark. We further present and discuss examples of their usage.

EFLBench, BUT4Reuse and the used feature location techniques are available at `http://github.com/but4reuse/but4reuse/wiki/Benchmarks`.

The rest of the paper is structured as follows: In Section 4 we present Eclipse as a case study subject and in Section 5 we present the EFLBench framework. Section 6 presents different feature location techniques and the results of EFLBench usage in the official Eclipse releases. Section 7 presents the strategies for automatic generation of Eclipse variants and examples of their usage. Finally, Section 9 concludes and presents future work.

2. Background on feature location in feature-based variants

Features are the entities used to distinguish the variants of an SPL. In this context, a feature is defined as "a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems" [12]. This definition is very general and open to interpretation so one recurrent challenge in implementing SPLs is deciding the granularity that the features will have at the implementation level [13]. Coarse granularity (e.g., components or plugins [14, 15, 16]) makes easier the maintenance of the SPL while fine granularity (e.g., source code classes or code fragments [17, 18]) might complicate the development and maintenance of the SPL. This way, there are very diverse scenarios regarding the granularity of the reusable assets in SPL.

Depending on the granularity, feature location can focus on code fragments in the case of source code [19, 20, 21, 22], model fragments in the context of models [23] or software components in software architectures [15, 14, 16]. Therefore, existing techniques are composed of two phases: An *abstraction* phase, where the different artefact variants are abstracted, and the *location* phase where algorithms analyse or compare the different product variants to obtain the implementation elements associated to each feature. Despite these two phases, the existing works differ in:

- *The way the product variants are abstracted and represented.* Indeed, each approach uses a specific formalism to represent product variants. For example, AST nodes for source code [20], model elements to represent model variants [7] or plugins in software architectures [15]. Some use fine granularity using AST nodes that cover all source code statements while others use purposely a bigger granularity using object-oriented building elements [22], like Salman *et al.* that only consider classes [24].

- *The proposed algorithms.* Each approach proposes its own algorithm to analyse product variants and identify the groups of elements that are related to features. Rubin *et al.* [7] and Wesley *et al.* [8] conducted surveys about the state-of-the-art in this domain. They showed the variety of techniques and application domains. For instance, Fischer *et al.* used a static analysis algorithm [20]. Other approaches use techniques from the field of Information Retrieval (IR). Xue *et al.* [25] and Salman *et al.* [26] proposed the use of Formal Concept Analysis (FCA) [27] to group implementation elements in blocks and then, in a second step, the IR technique Latent Semantic Indexing (LSI) [28] to map between these blocks and the features. Salman *et al.* used hierarchical clustering to perform this second step [24].

Figure 1 illustrates the feature location task in feature-based variants. In the upper half we illustrate the abstraction phase and in the lower half we illustrate the location phase. We present a set of variants (four circumferences in the figure) and their implementation elements (rhombuses). For each of the variants, we also have the information of which features are implemented. Feature location techniques in software families use to assume that feature presence or absence in the product variants is known upfront [20]. For example, Variant 1 implements F1, F2 and F3 while Variant 2 implements F1 and F3 as well but not F2. Despite that we know if a feature is implemented in a variant, we do not know the implementation elements associated to it. Therefore, the feature location algorithm takes the information of all the variants (features and implementation elements) and decide, for each feature, which are the associated implementation elements as shown at the bottom of Figure 1.

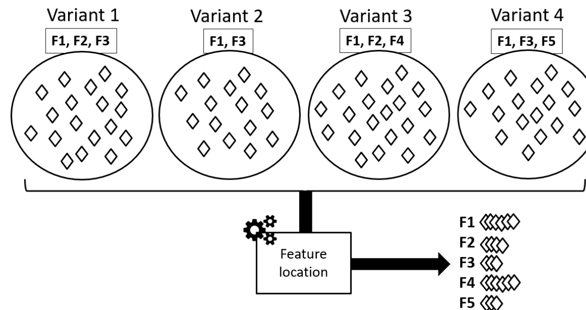


Figure 1: Feature location in feature-based variants.

3. Related work

Benchmarks: In SPL Engineering (SPLE), several benchmarks and common test subjects have been proposed. Lopez-Herrejon *et al.* proposed evaluating SPL technologies on a common SPL, a Graph Product Line [29], whose variability features are familiar to any computer engineer. The same authors proposed a benchmark for combinatorial interaction testing techniques for SPLs [30]. Also, automated FM analysis has a long history in SPLE research [31]. FAMA is a tool for feature model analysis that allows to include new reasoners and new reasoning operators [32]. Taking as input these reasoners, the BeTTy framework [33], built on top of FAMA, is able to benchmark the reasoners to highlight the advantages and shortcomings of different analysis approaches.

Feature location on software families is also becoming more mature with a relevant proliferation of techniques [7, 8]. Therefore, benchmarking frameworks to support the evolution of this field are in need. Different case studies have been used for evaluating feature location in software families [8]. For instance, ArgoUML variants have been extensively used [17]. However, none of the presented case studies have been proposed as a benchmark except the variants of the Linux kernel by Xing *et al.* [18]. This benchmark considers twelve variants of the Linux kernel from which a ground truth is extracted with the traceability of more than two thousands features to code parts. The Linux kernel benchmark can be considered as complementary to advance feature location research because EFLBench a) maps to a project that is plugin-based, while Linux considers C code, and b) the characteristics of the natural language terminology is different from the Linux kernel terminology. This last point is important because techniques based on information retrieval techniques should be evaluated in different case studies. EFLBench is integrated with BUT4Reuse which is extensible for feature location techniques making easier to control and reproduce the settings of the studied techniques.

Feature location: Liu *et al.* and Kästner *et al.* among others proposed to identify feature information from a single product [34, 35]. There are SPL adoption scenarios where the SPL wants to be extracted from a single product by separating its features. However, in this paper we concentrate on the case of several artefact variants.

Feature location has been investigated in other software engineering fields such as in maintenance (e.g., determining relevant elements for a modification task [1, 2]). These techniques have been also used in extractive SPL adoption. Alves *et al.*, in a case study of commercial mobile game variants [36], instead of using static comparison techniques, located the implementation elements of the known features through concern graphs [2]. Kästner *et al.* proposed a semi-automatic approach for feature location in single systems where, as input, the domain expert manually needs to point the system to relevant fragments of an artefact with respect to a feature [35]. Then, the approach automatically expands this user selection using information about element dependencies.

Block identification as a previous step to locate features: To distinguish features and their associated elements, researchers have proposed to

analyse and compare artefact variants for the identification of their common and variable parts [37, 38, 21, 39, 20]. We refer to each of such distinguishable parts as a *block*. A block is a set of implementation elements of the artefact variants that are relevant for the targeted mining task. Examples of existing techniques to identify blocks are based on static analysis, dynamic analysis or information retrieval techniques [8]. Independently of the technique or artefact type, a block is an intermediary abstraction representing a candidate set of elements that might implement a feature. In the literature on feature location from artefact variants, we can find the same concept of *blocks* with different names. Rubin *et al.* call them *parts*, *regions*, or *diff-sets* alluding to the technique used to retrieve them [39]. Other example of generic names are *modules* by Méndez-Acuña *et al.* [40] or *clusters* by Yang *et al.* [37] and Araar *et al.* [41]. Other employed terminology is less generic and they specifically refers to the concrete artefact types that they are dealing with. Linsbauer *et al.* [42] and Salman *et al.* [26] refer to blocks as potential *feature-to-code mappings* or *traces*. AL-msie'deen *et al.* call them *object-oriented building elements sets* [38] and *atomic blocks* [43]. Each calculated block cannot be directly considered the implementation of a feature. In these approaches they propose heuristics or they consider that the final mapping is a manual process based on domain expertise.

4. The Eclipse family of integrated development environments

The Eclipse community, with the support of the Eclipse Foundation, provides integrated development environments (IDEs) targeting different developer profiles. The IDEs cover the development needs of *Java*, *C/C++*, *JavaEE*, *Scout*, *Domain-Specific Languages*, *Modeling*, *Rich Client Platforms*, *Remote Applications Platforms*, *Testing*, *Reporting*, *Parallel Applications* or for *Mobile Applications*. Following Eclipse terminology, each of the customized Eclipse IDEs is called an **Eclipse package**. To avoid confusion with Java packages, we will refer to Eclipse packages as **variants** in the rest of the paper.

As the Eclipse project evolves over time, new variants appear and some other ones disappear depending on the interest and needs of the community. For instance, in 2012, one variant for *Automotive Software* developers appeared and, recently, in 2016, another variant appeared for *Android mobile applications development*. The Eclipse Packaging Project (EPP) is the technical responsible for creating entry level downloads based on defined user profiles.

Continuing with Eclipse terminology, a *simultaneous release* (**release** hereafter) is a set of variants which are public under the supervision of the Eclipse Foundation. Every year, there is one main release, in June, which is followed by two service releases for maintenance purposes: SR1 and SR2 usually around September and February. For each release, the platform version changes and traditionally celestial bodies are used to name the releases, for example Luna for version 4.4 and Mars for version 4.5.

The variants present variation depending on the included and not-included **features**. For example, Eclipse variant for Testers is the only one including

the Jubula Functional Testing features. On the contrary, other features like the Java Development tools are shared by most of the variants. There are also common features for all the variants, like the Equinox features that implement the core functionality of the Eclipse architecture. The online documentation of each release provides high-level information on the features that each variant provides [1].

It is important to mention that in this work we are not interested in the variation among the releases (e.g., version 4.4 and 4.5, or version 4.4 SR1 and 4.4 SR2), known as *variation in time*. We focus on the variation of the different variants of a given release, known as *variation in space*, which is expressed in terms of included and not-included features. Each variant is different in order to support the needs of the targeted developer profile by including only the appropriate features.

Eclipse is feature-oriented and based on **plugins**. Each feature consists of a set of plugins that are the actual implementation of the feature. Table 1 shows an example of feature with four plugins as implementation elements that, if included in an Eclipse variant, adds support for the Concurrent Versioning System (CVS). At technical level, the actual features of a variant can be found within a folder called *features* containing meta-information regarding the included features and the list of plugins associated to each. A feature has an id, a name and a description as defined by the feature providers of the Eclipse community. A plugin has an id and a name defined by the plugin providers, but it does not have a description.

Table 1: Eclipse feature example. The Eclipse CVS Client feature and its associated plugins.

Feature	
id: org.eclipse.cvs	
name: Eclipse CVS Client	
description: Eclipse CVS Client (binary runtime and user documentation).	
Plugin id	**Plugin name**
org.eclipse.cvs	Eclipse CVS Client
org.eclipse.team.cvs.core	CVS Team Provider Core
org.eclipse.team.cvs.ssh2	CVS SSH2
org.eclipse.team.cvs.ui	CVS Team Provider UI

Table 2 presents data regarding the evolution of the Eclipse releases over the years. In particular, it presents the total number of variants, features and plugins per release. To illustrate the distribution of variants and features, Figure 2 depicts a matrix of the different Eclipse Kepler SR2 variants where a black box denotes the presence of a feature (horizontal axis) in a variant (vertical axis). We observe that some features are present in all the variants while others are

[1] High-level comparison of Eclipse variants of the latest release: `https://eclipse.org/downloads/compare.php`

Table 2: Eclipse releases and their number of variants, features and plugins.

Year	Release	Variants	Features	Plugins
2008	Europa Winter	4	91	484
2009	Ganymede SR2	7	291	1,290
2010	Galileo SR2	10	341	1,658
2011	Helios SR2	12	320	1,508
2012	Indigo SR2	12	347	1,725
2013	Juno SR2	13	406	2,008
2014	Kepler SR2	12	437	2,043
2015	Luna SR2	13	533	2,377

specific to only few variants. The 437 features are alphabetically ordered by their id. For instance, the feature *Eclipse CVS Client*, tagged in the figure, is present in all variants except in the *Automotive Software* variant.

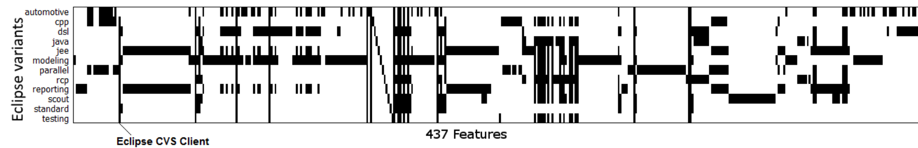


Figure 2: Eclipse Kepler SR2 variants and a mapping to their 437 features. For example, Eclipse CVS Client is present in all variants except in the automotive variant.

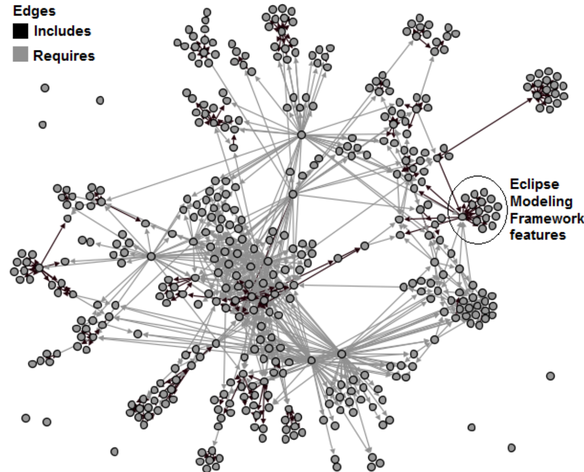


Figure 3: Feature dependencies in the Eclipse Kepler SR2 variants.

Features have dependencies among them: *Includes* is the Eclipse terminology to define subfeatures, and *Requires* means that there is a functional dependency

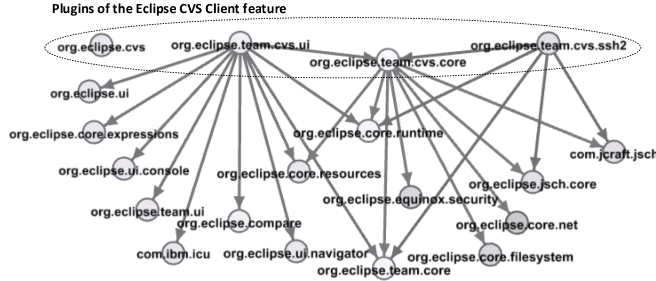


Figure 4: Plugin dependencies of the four plugins of the Eclipse CVS Client feature.

between the features. Figure 3 shows the dependencies between all the features of all variants in Eclipse Kepler SR2. We tagged some features and subfeatures of the Eclipse Modeling Framework to show cases of features that are strongly related. Functional dependencies are mainly motivated by the existence of dependencies between plugins of different features. In the Eclipse IDE family there is no *excludes* constraint between the features. Regarding plugin dependencies, they are explicitly declared in each plugin meta-data. Figure 4 shows a small excerpt of the dependency connections of the 2043 plugins of Eclipse Kepler SR2. Concretely, the excerpt shows the dependencies of the four CVS plugins presented in Table 1.

4.1. Reasons to consider Eclipse for benchmarking

We present characteristics of Eclipse variants that make the case study interesting for a feature location benchmark:

Ground truth available: The Eclipse case study fulfils the requirement, mentioned in Section 1, of providing the needed data to be used as ground truth. This ground truth can be extracted from features meta-information. Despite that the granularity of the implementation elements (plugins) is coarse if we compare it with source code AST nodes, the number of plugins is still reasonably high. In Eclipse Kepler SR2, the total amount of unique plugins is 2043 with an average of 609 plugins per Eclipse variant and a standard deviation of 192.

Challenging: The relation between the number of available variants in the different Eclipse releases (around 12) and the number of different features (more than 500 in the latest release) is not balanced. This makes the Eclipse case study challenging for techniques based only in static comparison (e.g., interdependent elements or FCA) because they will probably identify few “big” blocks containing implementation elements belonging to a lot of features. The number of available product variants has been shown to be an important factor for feature location techniques [20].

Friendly for information retrieval and dependency analysis: Eclipse feature and plugin providers have created their own natural language vocabulary. The feature and plugin names (and the description in the case of the

features) can be categorized as meaningful names [7] enabling the use of several IR techniques. Also, the dependencies between features and dependencies between implementation elements have been used in feature location techniques. For example, in source code, program dependence analysis has been used by exploiting program dependence graphs [44]. Acher *et al.* also leveraged architecture and plugin dependencies [15]. As presented in previous section, Eclipse also has dependencies between features and dependencies between plugins enabling their exploitation during feature location.

Noisy: There are properties that can be considered as "noise" that are common in real scenarios. Some of them can be considered as non-conformities in feature specification [45]. A case study without "noise" should be considered as an optimistic case study. In Eclipse Kepler SR2, 8 plugins do not have a name, and different plugins from the same feature are named exactly the same. There are also 177 plugins associated to more than one feature. Thereby the features' plugin sets are not completely disjoint. These plugins are mostly related to libraries for common functionalities which were not included as required plugins but as a part of the feature itself. In addition, 40 plugins present in some of the variants are not declared in any feature. Also, in few cases, feature versions are different among variants of the same release.

Friendly for customizable benchmark generation: The fact that Eclipse releases contain few variants can be seen as a limitation for benchmarking in other desired scenarios with larger amount of variants. For example, it will be desired to show the relation between the results of the technique and the number of considered variants. Apart from the official releases, software engineering practitioners have created their own Eclipse variants. Therefore, researchers can use their own variants or create variants with specific characteristics. In addition, the plugin-based architecture of Eclipse allows to implement automatic generators of Eclipse variants as we present later in Section 7.

Similar experiences exist: Analysing plugin-based or component-based software system families to leverage their variability has been shown in previous works [15, 14, 16]. For instance, experiences in an industrial case study were reported by Grünbacher *et al.* where they performed manual feature location in Eclipse variants to extract an SPL involving more than 20 Eclipse customizations per year [14].

5. EFLBench: Eclipse Feature Location Benchmarking framework

EFLBench is aimed to be used with any set of Eclipse variants including variants with features that are not part of any official release. Figure 5 illustrates, at the top, the phase for constructing the benchmark and, at the bottom part, the phase for using it. The following subsections provide more details on the two phases.

In Section 2 we presented the principles for feature location in feature-based systems. EFLBench follows these assumptions for a feature location task and provide the following **inputs** for the feature location technique:

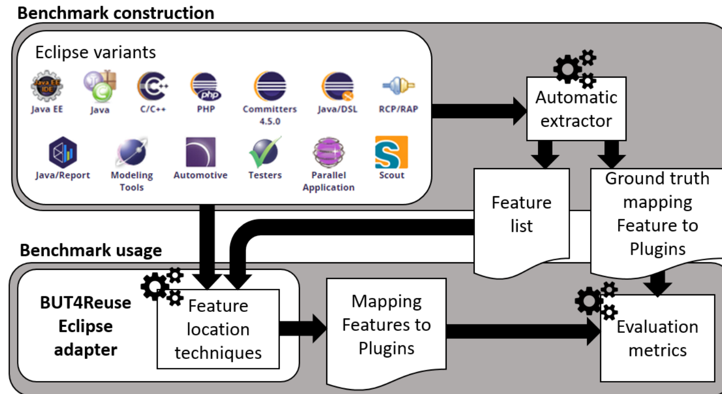


Figure 5: EFLBench: Eclipse variants as benchmark for feature location.

- The feature names and descriptions
- For each feature, the list of variants where it was included
- The dependencies between features
- The plugin names
- The dependencies between plugins

5.1. Benchmark construction

The benchmark construction phase takes as input the Eclipse variants and automatically produce two outputs, 1) a Feature list with information about each feature name, description and the list of variants where it was present, and 2) a ground truth with the mapping between the features and the implementation elements which are the plugins.

We implemented an automatic extractor of features information. The information is available in the file feature.xml of each feature so it was easy to automatically get the metadata (name, description, dependencies etc.) corresponding to all features. The implementation elements of a feature are those plugins that are directly associated to this feature. From the 437 features of the Eclipse Kepler SR2, each one has an average of 5.23 plugins associated with, and a standard deviation of 9.67 plugins. There is one outlier with 119 plugins which is the feature *BIRT Framework* included in the Reporting variant. From the 437 features, there are 19 features that do not contain any plugin, so they are considered *abstract* features which are created just for grouping other features. For example, the abstract feature *UML2 Extender SDK* (Software Development Kit) includes the features *UML2 End User Features*, *Source for UML2 End User Features*, *UML2 Documentation* and *UML2 Examples*.

Reproducibility can become easier by using benchmarks and common frameworks that launch and compare different techniques [10]. This practice, allows a valid performance comparison with all the implemented and future techniques. We integrated EFLBench and its automatic extractor in BUT4Reuse.

5.2. Benchmark usage

Once the benchmark is constructed, at the bottom of Figure 5 we illustrate how it can be used through BUT4Reuse where feature location techniques can be integrated. The Eclipse adapter [11] is responsible for the variant abstraction phase. During the product abstraction phase, the implemented Eclipse adapter decomposes any Eclipse installation in a set of plugins by visiting and analysing the Eclipse variant file structure. The plugin elements contain information about their id, name as well as their dependency to other plugin elements. This will be followed by the launch of the targeted feature location techniques which takes as input the feature list and the Eclipse variants (excluding the *features* folder). The feature location technique produces a mapping between features and plugins that can be evaluated against the ground truth obtained in the benchmark construction phase. Concretely, EFLBench calculates the *precision* and *recall* which are classical evaluation metrics in IR studies (e.g., [24]).

We explain precision and recall, two metrics that complement each other, in the context of EFLBench. A feature location technique assigns a set of plugins to each feature. In this set, there can be some plugins that are actually correct according to the ground truth. Those are *true positives* (TP). TPs are also referred to as *hit*. On the set of plugins retrieved by the feature location technique for each feature, there can be other plugins which do not belong to the feature. Those are *false positives* (FP) which are also referred to as *false alarms*. Precision is the percentage of correctly retrieved plugins relative to the total of retrieved plugins by the feature location technique. A precision of 100% means that all retrieved plugins are contained in the ground truth set and that no false alarm plugins were included. The formula of precision is shown in Equation 1.

$$precision = \frac{TP}{TP + FP} = \frac{plugins\,hit}{plugins\,hit\, +\, plugins\,false\,alarm} \tag{1}$$

According to the ground truth there can be some plugins that are not included in the retrieved set, meaning that they are *miss*. Those plugins are *false negatives* (FN). Recall is the percentage of correctly retrieved plugins from the set of the ground truth. A recall of 100% means that all the plugins of the ground truth were assigned to the feature. The formula of recall is shown in Equation 2.

$$recall = \frac{TP}{TP + FN} = \frac{plugins\,hit}{plugins\,hit\, +\, plugins\,miss} \tag{2}$$

Precision and recall are calculated for each feature. In order to have a global result of the precision and recall we use the mean of all the features. Finally, BUT4Reuse reports the *time* spent for the feature location technique. With this information, the time performance of different techniques can be compared.

6. Examples of EFLBench usage in Eclipse releases

This section aims at presenting the possibilities of EFLBench by benchmarking four feature location techniques in official Eclipse releases. For the four techniques we use Formal Concept Analysis (FCA) [27] as a first step for block identification and the four feature location techniques are Strict Feature Specific (SFS), SFS+ST, SFS+TF, SFS+TFIDF which we detail in next subsection before presenting the results.

6.1. Background on techniques used in the examples

FCA [27] uses a *formal context* as input and groups elements that share common attributes. The entities of the formal context are the variants, and the attributes (binary attributes) are the presence or absence of each of the elements in each variant. With this input, FCA discovers a set of *concepts* and the concepts containing at least one element are considered as a block for the feature location task. Figure 6 illustrates FCA. The identified blocks correspond to the different intersections from the input artefact variants. A detailed explanation about FCA formalism in the same context of extractive SPL adoption can be found in Al-Msie'deen *et al.* [22] and Shatnawi *et al.* [16]. At technical level, we implemented FCA for block identification using Galatea. [2]

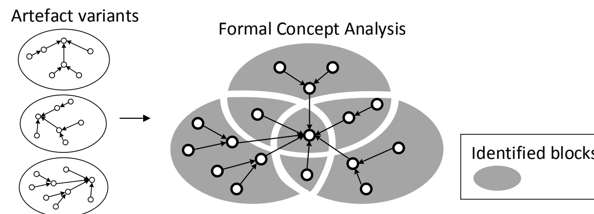


Figure 6: Illustration of block identification with Formal Concept Analysis.

SFS is a feature location technique that follows two assumptions: A feature is located in a block when 1) the block *always* appears in the artefacts that implements this feature and 2) the block *never* appears in any artefact that does not implement this feature. The principles of this feature location technique are similar to locating distinguishing features using diff sets [19].

Natural Language Processing (NLP) techniques: In SFS+ST, SFS+TF, SFS+TFIDF, where we use IR and NLP, we do not make use of the feature or plugin ids. In order to extract the meaningful words from both features (name and description) and elements (plugin names), we used two well established techniques in the IR field.

[2] Galatea Formal Concept Analysis library: `https://github.com/jrfaller/galatea`

- Parts-of-speech tags remover: These techniques analyse and tag words depending on their role in the text. The objective is to filter and keep only the potentially relevant words. For example, conjunctions (e.g., "and"), articles (e.g., "the") or prepositions (e.g., "in") are frequent and may not add relevant information. As an example, we consider the following feature name and description: "*Eclipse Scout Project. Eclipse Scout is ~~a~~ business application framework ~~that~~ supports desktop, web ~~and~~ mobile frontends. ~~This~~ feature contains ~~the~~ Scout core runtime components.*". We apply Part-of-Speech Tagger techniques using OpenNLP [46].

- Stemming: This technique reduces the words to their root. The objective is to unify words not to consider them as unrelated. For instance, "playing" will be considered as stemming from "play" and "tools" from "tool". Instead of keeping the root, we keep the word with greater number of occurrences to replace the involved words. As example, in the Graphiti feature name and description we find "[...]*Graphiti supports the fast and easy creation of unified **graphical** tools, which can **graphically** display[...]*" so graphical and graphically is considered the same word as their shared stem is *graphic*. Regarding the implementation, we used the Snowball stemmer [47].

SFS and Shared term: The intuition behind this technique is first to group features and blocks with SFS and then apply a "search" of the feature's words within the elements of the block to discard elements that may be completely unrelated to the feature. For each association between feature and block, we keep, for this feature, only the elements of the block that have at least one meaningful word shared with the feature. That means that we keep the elements whose *term frequency* (tf) between feature and element (featureElementTF) is greater than zero. For clarification, featureElementTF is defined in Equation 3 being f the feature, e the element and `tf` a method that just counts the number of times a given term appears in a given list of terms.

$$featureElementTF(f, e) = \sum_{term_i \in e.terms} tf(term_i, f.terms) \qquad (3)$$

Figure 7 illustrates, on the left side, how for a given feature, we have associated words and how, from a block obtained with SFS, we discard elements that do not share any word with the feature.

SFS and Term frequency: After employing SFS, this technique is based on the idea that all the features assigned to a block *compete* for the block elements. The feature (or features in case of drawback) with higher *featureElementTF* will keep the elements while the other features will not consider this element as part of it. Figure 7 illustrates this technique in the center of the figure. Three features compete for the elements of a block obtained with SFS, and the assignation is made by calculating the `tf` between each element and the features. That means that, for each element, the feature with higher `tf` with respect to the element will be the only feature that is mapped to this element.

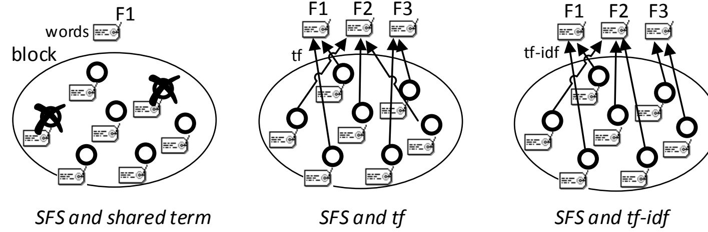


Figure 7: Three different feature location techniques using SFS and term frequency.

SFS and tf-idf: Figure 7, on the right side, illustrates this technique. SFS is applied and then the features also compete, in this case, for the elements of the block but a different weight is used for each word of the feature. This weight (or score) is calculated through the *term frequency - inverse document frequency* (`tf-idf`) value of the set of features that are competing. `tf-idf` is a well known technique in IR [48]. `tf` is a metric consisting in giving more relevance to the terms appearing with more frequency in a document d. When dealing with a set D of documents $d_1, ..., d_n$, *term frequency-inverse document frequency* (`tf-idf`) is another metric used in IR [48]. For a document d, `tf-idf` penalizes common terms that appear across most of the documents in D and emphasizes those terms that are more specific to d. There are different formulas to calculate them. In this work, we used the formulas presented in Equation 4, where we use *raw term frequency* (`tf`) which is calculated counting the occurrences of a given term in a document, *inverse document frequency* (`idf`) which measures how much rare or common a term is across all the documents using a logarithmic scale and, finally, `tf-idf` uses `tf` multiplied by `idf` to penalize or encourage a term depending on its occurrence across D. In our context, the idea is that words appearing more frequently through the features may not be as important as less frequent words.

$$\begin{aligned} tf(term_i, d) &= f_{term_i,d} \\ idf(term_i, D) &= \log\left(\frac{|D|}{|\{d \in D : term_i \in d\}|}\right) \\ \mathit{tf\text{-}idf}(term_i, d, D) &= tf(term_i, d.terms) \times idf(term_i, D) \end{aligned} \quad (4)$$

Given that `tf-idf` is used in SFS+TFIDF, we illustrate it in the context of Eclipse features. For example “Core”, “Client” or “Documentation” are more frequent words across features but “CVS” or “BIRT”, being less frequent, are probably more relevant, informative or discriminating.

6.2. Results in Eclipse releases

We used the benchmark created with each of the Eclipse releases presented in Table 2. The experiments were launched using BUT4Reuse (commit *ce3a002*)

which contains the presented feature location techniques. Detailed instructions for reproducibility are available [3]. We used a laptop Dell Latitude E6330 with a processor Intel(R) Core(TM) i7-3540M CPU@3.00GHz with 8GB RAM and Windows 7 64-bit.

After using the benchmark, we obtained the results shown in Table 3. *Precision* and *Recall* are the mean of all the features as discussed at the end of Section 5.2. The results in terms of precision are not satisfactory in the presented feature location techniques. This suggests that the case study is challenging. Also, we noticed that there are no relevant differences in the results of these techniques among the different Eclipse releases. As discussed before, given the small number of Eclipse variants under consideration, FCA is able to distinguish blocks which may actually correspond to a high number of features. For example, all the plugins corresponding specifically to the Eclipse Modeling variant, will be grouped in one block while many features are involved. Despite that these techniques are used in feature location of feature-based variants we provide these results to be used as baselines to motivate the search of more accurate feature location techniques and to show that the benchmark is appropriate to advance the research in this field.

Another example, in Eclipse Kepler SR2, FCA-based block identification identifies 60 blocks with an average of 34 plugins per block and a standard deviation of 54 plugins. In Eclipse Europa Winter, with only 4 variants, only 6 blocks are identified with an average of 80 plugins each and a standard deviation of 81. Given the low number of Eclipse variants, FCA identifies a low number of blocks. The number of blocks is specially low if we compare it with the actual number of features that we aim to locate (e.g., 60 blocks in Kepler SR2 against its 437 features). The higher the number of Eclipse variants, the more likely FCA will be able to distinguish different blocks.

Table 3: Precision (Prec) and recall of the different feature location techniques.

	SFS		**SFS+ST**		**SFS+TF**		**SFS+TFIDF**	
Release	**Prec**	**Recall**	**Prec**	**Recall**	**Prec**	**Recall**	**Prec**	**Recall**
Europa Winter	6.51	99.33	11.11	85.71	12.43	58.69	13.07	53.72
Ganymede SR2	5.13	97.33	10.36	87.72	11.65	64.31	12.80	52.70
Galileo SR2	7.13	93.39	10.92	82.01	11.82	60.50	12.45	53.51
Helios SR2	9.70	91.63	16.04	80.98	25.97	63.70	29.46	58.39
Indigo SR2	9.58	92.80	15.72	82.63	19.79	59.72	22.86	57.57
Juno SR2	10.83	91.41	19.08	81.75	25.97	61.92	24.89	60.82
Kepler SR2	9.53	91.14	16.51	83.82	26.38	62.66	26.86	57.15
Luna SR2	7.72	89.82	13.87	82.72	22.72	56.67	23.73	51.31
Mean	*8.26*	*93.35*	*14.20*	*83.41*	*19.59*	*61.02*	*20.76*	*55.64*

The first location technique (FCA+SFS) does not assume meaningful names given that no IR technique is used. The features are located in the elements

[3] `https://github.com/but4reuse/but4reuse/wiki/Benchmarks`

of a whole block obtaining a high recall (few plugins missing). Eclipse feature names and descriptions are probably written by the same community of developers that create the plugins and decide their names. In the approaches using IR techniques, it was expected a higher increment of precision without a loss of recall but the results suggest that a certain divergence exists between the vocabulary used at feature level and at implementation level.

Regarding the time performance, Table 4 shows, in milliseconds, the time spent for the different releases. The *Adapt* column corresponds to the time to decompose the Eclipse variants into a set of plugin elements and get their information. This adaptation step heavily rely to access the file system and we obtain better time results after the second adaptation of the same Eclipse variant. The FCA time corresponds to the time for block identification. We consider Adapt and FCA as the preparation time. Then, the following columns show the time of the different feature location techniques. We can observe that the time performance is not a limitation of these techniques as they take a maximum of around half a minute.

Table 4: Time performance in milliseconds for feature location.

	Preparation		*Concrete techniques*			
Release	**Adapt**	**FCA**	**SFS**	**SFS+ST**	**SFS+TF**	**SFS+TFIDF**
Europa Winter	2,397	75	6	2,581	2,587	4,363
Ganymede SR2	7,568	741	56	11,861	11,657	23,253
Galileo SR2	10,832	1,328	107	17,990	17,726	35,236
Helios SR2	11,844	1,258	86	5,654	5,673	12,742
Indigo SR2	12,942	1,684	100	8,782	8,397	16,753
Juno SR2	16,775	2,757	197	7,365	7,496	14,002
Kepler SR2	16,786	2,793	173	8,586	8,776	16,073
Luna SR2	17,841	3,908	233	15,238	15,363	33,518
Mean	*12,123*	*1,818*	*120*	*9,757*	*9,709*	*19,493*

It is out of the scope of the EFLBench contribution to propose feature location techniques that could obtain better results in the presented cases. The objective is to present the benchmark usage showing that quick feedback from feature location techniques can be obtained in the Eclipse releases case studies. In addition, we provide empirical results of four feature location techniques that can be used as baseline.

7. Automatic and parametrizable generator of Eclipse variants

The main motivation for the generation of variants is that it enables to evaluate the feature location techniques in controlled settings. As shown in Table 2, the number of official variants of an Eclipse release amounts to around 12 Eclipse variants. In order to provide a framework for intensive evaluation of feature location techniques, cases with larger number of Eclipse variants are desired. In addition, a parametrizable number of variants could serve to analyse the results of the same feature location technique under different circumstances.

For instance, it is interesting to evaluate the same technique in cases with variants which are similar, or dissimilar, among them. Using the Jaccard similarity measure between pairs of variants [49, 50] (calculated as the size of the intersection of the selected features divided by the size of the union) and considering the official releases, we observe that the average similarity ranges from the 22% of Ganymede SR2 to the 27% of Galileo SR2, with an average of 25% for the eight presented releases. Therefore, these families are homogeneous in terms of the average similarity between variants. However, it is desired to experiment in other settings to evaluate this factor in the different techniques.

It is not evident where to find real Eclipse configurations and how to group them to satisfy certain desired characteristics, therefore we extended our framework with the generation of variants enabling the possibility to create several settings regarding the number of variants and the similarity among them. We extended the benchmark construction phase of EFLBench with an automatic and parametrizable generator of Eclipse variants to construct benchmarks with tailored characteristics. The approach consists in automatically creating variants taking as input a user-specified Eclipse variant.

We agree that generated variants are synthetic variants which can be seen as non representative variants of realistic cases (i.e., we cannot validate if the set of features makes sense for a real development scenario). For using EFLBench with realistic variants we should rely on the official Eclipse releases as we presented in Section 6. For the generated variants we can only guarantee the following two characteristics.

- Feature constraints are respected (i.e., dependencies of the features)
- The Eclipse variant can be executed.

Figure 8 illustrates the benchmark construction phase using the automatic generation of Eclipse variants. First, as shown on the upper left side of the figure, we take as input an Eclipse variant to extract its features and feature constraints. These features and constraints define a configuration space in the sense that, by deselecting features, we can still have valid Eclipse configurations (i.e., all the feature constraints are satisfied). Then, we leverage this configuration space to select a set of configurations. The automatic selection of configurations is parametrized by a given strategy, thus, this step is extensible to different implementations. Below, we present three different strategies that we have implemented. Finally, once the set of configurations are selected, we implemented an automatic method to construct the variants through the input Eclipse and the feature configurations. The constructed variants are created for preparing the benchmark construction but, if desired, given that constraints are respected, they can be executed in the same way as the variants in Eclipse releases.

7.1. Strategies for the automatic selection of configurations

We implemented three strategies to select configurations from a set of features and constraints with the final objective to construct benchmarks presenting different characteristics. Apart from the input Eclipse, the three take as

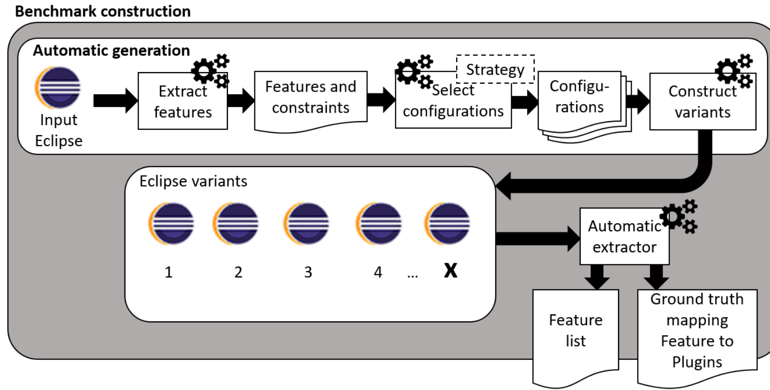


Figure 8: Automatic and parametrizable generation of Eclipse variants to construct a feature location benchmark. The use of different strategies in the step to select configurations enables to construct benchmarks exhibiting different characteristics.

input a user-specified number of variants (n) that want to be generated. We present the three strategies and then discuss their properties:

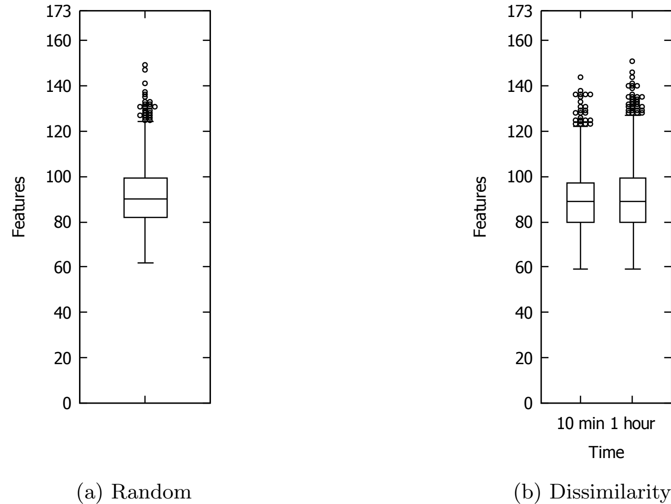
- *Random selection strategy*: In this strategy, we randomly select n configurations from the configuration space. The configuration space is the set of all possible valid configurations (those that satisfy all the constraints among features). Therefore, this strategy can be illustrated as repeating n times the selection of a random number from one to the size of the configuration space, and then taking the feature configuration associated to this number. The selection of random valid configurations, taking as input features and their constraints, is implemented through a functionality offered by the PLEDGE library (Product Line Editor and tests Generation tool) [51]. We used the PLEDGE tool as a black box library as it fitted our needs and that had already proven useful in other cases of randomly selecting configurations in the way we have described. PLEDGE internally relies on a boolean satisfiability problem solver (SAT solver) [52, 53].

- *Random selection strategy trying to maximize dissimilarity*: This strategy aims to obtain a set of n configurations that maximize their global dissimilarity. That means that an optimization algorithm explores the configuration space trying to find the set of n configurations from the configuration space that are more different among them. For this we use again the available PLEDGE functionality. First, PLEDGE selects n random configurations and then, they evolve over time by performing mutations. Concretely, it applies a search-based approach guided by a fitness function that tries to identify the most dissimilar configurations based on the Jaccard distance [49, 50]. The best solution found (the set of n configurations) at time t is returned as result. This strategy demands to select the time (t) allocated to the search-based algorithm. Once the allowed time

is over, the set of configurations are obtained.

- *Percentage-based random selection strategy*: This strategy consists of two steps. First, we ignore the constraints and we go through the feature list deciding if we select or not each feature. This is automated by a user-specified percentage (p) defining the chances of the features of being selected. Second, once some features are randomly selected, we need to guarantee that the feature constraints are satisfied. We may have included a feature that requires another one that was not included. Therefore, we *repair* the configuration including the missing features until obtaining a valid configuration. This strategy does not use PLEDGE. Since Eclipse features only provide dependency constraints, satisfying those constraints using the mentioned repair approach is trivial and no SAT solver is needed.

The three algorithms for the strategies that we have presented have stochastic components. In the following paragraphs we show the characteristics that we can be expected from each of them based on empirical data of their usage.



(a) Random (b) Dissimilarity

Figure 9: Different settings of the first two strategies for selecting configurations taking as input the features and constraints extracted from the Modeling variant of the Eclipse Kepler SR2. Each boxplot shows the number of features in the selection of 1000 configurations.

Using as input the Modeling variant of Eclipse Kepler SR2, Figures 9 and 10 show, in the vertical axis, the number of features in 1000 automatically selected configurations using the presented strategies. The total number of features of the input Eclipse variant is 173 corresponding to the maximum value. Considering the feature constraints, the configuration space exceeds one million configurations. In the case of the random and dissimilarity strategies, as shown in Figures 9a and 9b, we can observe that only some outlier configurations reach a large number of selected features. Given that the dissimilarity strategy

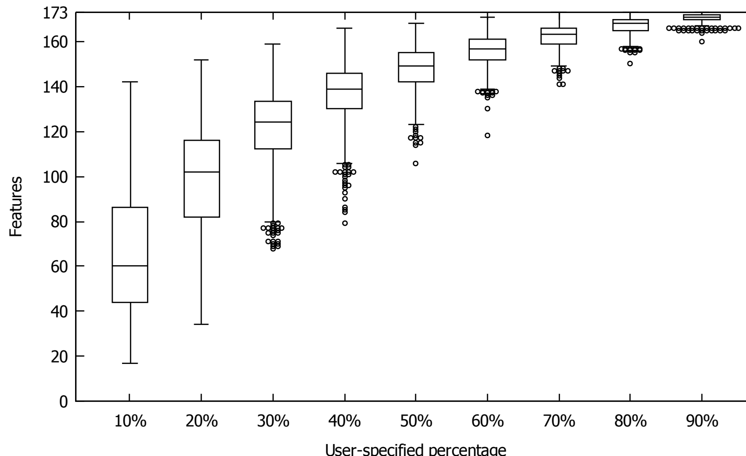


Figure 10: Different settings of the Percentage-based random selection strategy for selecting configurations taking as input the features and constraints extracted from the Modeling variant of the Eclipse Kepler SR2. Each boxplot shows the number of features in the selection of 1000 configurations.

depends on the number of desired variants to generate, we repeated the process with different number of configurations (not only 1000) obtaining analogous results. We also observed that the time allowed for the search-based algorithm did not affect the number of selected features, at least from 10 minutes to 1 hour as shown in Figure 9b. On the contrary, in Figure 10, we can observe how the user-specified percentage has an impact in the median of selected features. For example, using the random strategy, we should expect variants with around 50% of the features selected from the input Eclipse. On the contrary, if we select percentage-based random selection with 90% of user-specified percentage, we should expect variants with almost all the features selected from the input Eclipse.

Larger percentages using the percentage-based random selection allow to obtain configurations with a larger number of selected features and, therefore, there will be fewer chances to obtain dissimilar variants using this strategy compared to the ones using random selection. Empirical studies of Henard *et al.* showed that dissimilar configurations exhibit interesting properties in terms of pairwise coverage [50]. Pairwise coverage measure the coverage of all possible discrete combinations of features. The first and second strategy can be used to evaluate how a feature location technique behaves with dissimilar variants with high pairwise coverage. They also showed that the strategy of selecting random configurations from the configuration space, without the search-based step, already obtained a median of more than 90% of pairwise coverage in 120 FMs of moderate size (i.e., less than one thousand features). The third strategy, compared to the first two, allows to have more control over the total number of selected features per configuration.

7.2. Results using automatic generation of variants

We show examples of using the EFLBench strategies for automatic generation of Eclipse variants. We focus on discussing the results of evaluating the FCA+SFS feature location technique. This technique that first uses FCA and then SFS was presented in Section 6.1. As input for the random generation strategies, we use the Modeling variant of Eclipse Kepler SR2 which is the same used to illustrate the strategies for selecting configurations in Figures 9 and 10.

Using percentage-based random selection of features, we aim to empirically analyse whether the number of available variants has an impact on the FCA+SFS technique. First, we generated 100 variants using 40% as percentage for feature selection. By setting this percentage, the first 10 variants cover the 173 features which is the total number of features of the input Eclipse. This allows the construction of different benchmarking settings adding 10 variants each time while keeping the total number of possible features constant.

Table 5: Precision, recall and time measures in milliseconds of the FCA+SFS feature location technique in sets of randomly generated Eclipse variants using the percentage-based random strategy.

Percentage-based	**FCA+SFS**		*Time*	
random using 40%	**Precision**	**Recall**	**FCA**	**SFS**
10 variants	33.40	96.55	122	84
20 variants	47.91	96.02	415	320
30 variants	55.62	95.41	502	630
40 variants	58.60	95.41	1,268	905
50 variants	61.01	93.10	2,168	1,105
60 variants	62.57	90.73	2,455	1,382
70 variants	64.78	90.63	2,636	1,717
80 variants	65.40	90.02	4,137	4,049
90 variants	66.02	89.57	6,957	7,774
100 variants	66.02	89.57	7,515	7,251

Table 6: Precision, recall and time measures in milliseconds of the FCA+SFS feature location technique in sets of randomly generated Eclipse variants using the random strategy.

Random	**FCA+SFS**		*Time*	
	Precision	**Recall**	**FCA**	**SFS**
10 variants	72.83	86.33	328	190
20 variants	90.49	84.97	400	260
30 variants	91.81	84.97	451	394
40 variants	93.13	84.97	802	603
50 variants	93.13	84.97	1,122	905
60 variants	93.13	84.97	1,485	866
70 variants	93.13	84.97	1,878	2,961
80 variants	93.13	84.97	3,692	1,637
90 variants	93.80	84.97	4,539	1,567
100 variants	93.80	84.97	7,967	2,177

Table 5 shows the precision and recall obtained for FCA+SFS when considering different number of variants. We can observe how precision improves with the number of variants. From 10 to 20 variants, we have a precision improvement of around 15%. Beyond 30 variants, it seems that the included variants, with their feature combinations, are not adding more information that can be exploited by the FCA+SFS technique. As an extreme case, we can observe how we obtain the same precision with 90 and 100 variants even if we are including 10 more different variants. This non-linearity of the precision when we add more variants might seem counter-intuitive. However, it is related to the fact that adding more variants do not necessarily means that we are including new feature combinations that did not exist in the previous variants.

Regarding recall, independently of the number of variants we obtain very high levels of recall. It slightly decreases 7% from 10 to 100 variants, while precision increases, mainly because of the "noise" introduced by non-conformities in feature specification discussed in Section 4.1. Table 5 also presents time measures of one execution showing that the FCA+SFS technique scales correctly for 100 variants in this benchmark. Concretely, it took only around 15 seconds in total for FCA and SFS. If we include, as part of the feature location process, the time for adapting the variants using the Eclipse adapter (the *Adapt* time mentioned in Section 6), in the case of 100 variants it took 35 minutes which is still acceptable.

We used the same Modeling variant as input to generate 100 variants with the random selection strategy. As in the previous experiment, we keep the number of features constant given that 10 variants already cover the 173 features. Then, we calculate the results by incrementally adding another 10 variants. Table 6 shows the results of the same technique (FCA+SFS) in this new setting where we can observe that, with only 10 variants, we have 72.83% of precision. The result with 10 variants generated with this random selection strategy is better compared with the same number of variants generated through the percentage-based random selection which was 33.4% as shown in Table 5 (i.e., around 50% of difference in precision). Also, using 10 variants with the random strategy, the technique performs better than 100 variants with the percentage-based random selection (66.02% of precision). Then, starting with 20 variants we reach 90% precision and then from 40 to 80 variants it stays constant in 93.13%. This fact suggests again that including variants is not enough to increase the precision.

This result empirically suggests that the FCA+SFS feature location technique performs better when the variants are more dissimilar. We calculated the average Jaccard similarity between the variants using the two strategies: The random strategy creates groups of 10 variants with an average similarity of 41% while the percentage-based random selection (using 40%) has an average similarity of 73%. It seems that dissimilar configurations cover many more distinct pairs of features and thus make easier to locate the features.

It is worth to mention that the dissimilarity strategy obtained similar results as the ones presented in Table 6 which used the random strategy. In several runs, for 10 variants we obtain around 70% of precision while for 20 variants we already reach 90%. The average Jaccard similarity using the dissimilarity

strategy (with 10 minutes for the search-based step) is 37% which indicates that they are more dissimilar than the random strategy (41%). In this case, the marginal difference in terms of similarity (i.e., 4%) explains the small difference on the feature location results.

The presented examples are intended to show the capabilities of EFLBench in creating scenarios to compare the results of feature location techniques. Concretely, we have shown how to analyse the result 1) with different number of variants and 2) with the same number of variants but with different degrees of similarity. In the presented case of the FCA+SFS feature location technique, we provided empirical evidences that having more available variants do not necessarily means better results in precision. However, dissimilar variants is an important factor for obtaining higher levels of precision.

8. Threats to validity

The input for the feature location task presented in Section 5 might be considered few information if we compare it with concern location in maintenance tasks where it is a common practice to trace bug reports with the names and comments in the source code. However, in similar cases to our context of feature-based variants (e.g., the Linux-Kernel benchmark for feature location [18]), we can see that also feature names and descriptions are used as input. In this benchmark the feature location task is at a granularity of classes or code fragments, however in our case, it is at the coarse granularity of plugins where we only provide the plugin names as input. The description of features in Eclipse might be shorter than other kind of documents like bug reports, enhancement requests or other documentation such as requirements, however, this can be also seen as a challenging scenario to information retrieval techniques that will need to exploit other information (e.g., dependence graphs) to refine their results. In addition, EFLBench, being open-source, can be easily extended to integrate other sources of information to be used as input.

Regarding the granularity of Eclipse features, depending on the Eclipse community projects we can identify different levels of granularity (from coarse-grained to less coarse-grained ones). This is related to how they have decided to group the functionalities. Their separation enables us to create a ground truth that comes from the Eclipse community instead of manually defining a ground truth which will be difficult to validate. However, it is worthy to mention that subfeatures are not part of the ground truth. For example, the feature of the editor to support C++ development can be separated in several functionalities such as editor syntax highlighting, code-completion etc. which are not part of the EFLBench ground truth. The editor support for C++, even if we can consider it a coarse-grained feature, there are still many features related to C++ in the Eclipse variants. For example, in Eclipse Kepler variants we have "C/C++ Development Tools","Autotools support", "GCC Cross Compiler", "Berkeley UPC (Unified Parallel C) Toolchain Support", "C99 LR Parser", "UPC (Unified Parallel C) Support", "Memory View Enhancements" and more than ten

optional features related to C++. We agree that each of them could be internally separated in more features but the number of optional features, as it is, it is already large. In industrial cases dealing with Eclipse variants [14] they discuss that more fine-grained variability might be desired. For example, they also consider different setting values inside a plugin as a feature. However, in their case study with the Siemens VAI MSS tool [14] their analysis is only at the level of plugins as we propose in EFLBench.

In Eclipse variants we can find features that are not "conventional" functional features. For example, one feature is "Graphical Modeling Framework (GMF) Runtime" and another feature is "Graphical Modeling Framework (GMF) Runtime Source" which contains the source code documentation of the GMF Runtime. The latter can be certainly seen as a non conventional feature. However, in Eclipse Kepler, "GMF Runtime" is available in the Automotive and Modeling variants while "GMF Runtime Source" is only available in Modeling and not in Automotive. As another example, "Equinox p2 Core Function" is a feature that exists in all Eclipse variants, however, "Equinox p2 Core Function Source" is only available in DSL, Modeling, RCP, Scout and Standard, and not in Automotive, Cpp, Java, JEE, Parallel, Reporting and Testing variants. This indicates that the inclusion of this non conventional features in an Eclipse variant is performed in the same way as they do for conventional ones. If a feature is a distinguishable characteristic of a system that is relevant to some stakeholder then it seems that they differentiate between the users of the runtime and the plugin developers.

The use of automatically generated variants can be seen as a limitation to the validity of evaluating feature location techniques using these inputs. However, in the feature location literature we find several cases where the variants are generated from an existing SPL [8]. For example, ArgoUML [17], the most used case study in feature location [8] was a single product which was reengineered as an SPL by decomposing its features [17]. The ArgoUML SPL is able to derive 256 variants but only around ten are selected for evaluating feature location techniques. Our random generation is based on the same principles used in ArgoUML. In our case, we take as input an Eclipse variant and we decompose it also in its features. Then, we select features using a given strategy to create the variants. Deriving variants from an existing SPL is a common practice in our research community as it is a way to have a ground truth to compare the results of the techniques (i.e., the mapping between features and implementation elements are known). This comes at the price of using "synthetic" variants which are valid regarding feature constraints but that can represent non realistic variants (i.e., we cannot validate if they can respond to real customer requirements). Apart from using realistic variants of the official Eclipse releases, several executions of the random generation approaches can provide complementary insights about the feature location techniques.

9. Conclusions

We have presented EFLBench, a framework and a benchmark for supporting research on feature location in artefact variants. Existing and future techniques dealing with this activity in extractive SPL adoption can find a challenging playground which is directly reproducible. The benchmark can be constructed from any set of Eclipse variants from which the ground truth is extracted. We have shown examples of its usage with the Eclipse variants of the official releases for analysing four different feature location techniques. We also provide automatic generation of Eclipse variants using three strategies to support the creation of different benchmarking scenarios. We discussed the evaluation of one of the feature location techniques using randomly generated sets of Eclipse variants. We provided evidences that the number of variants and the similarity among them are important factors for feature location techniques.

We plan to use the benchmark in order to evaluate existing and innovative feature location techniques while also encouraging the research community on using it as part of their evaluation. In order to extend our framework, there is interest in mining software repositories, forums and issue trackers to identify real configurations of Eclipse from practitioners beyond the official releases. Also, given the high proliferation of feature location techniques, meta-techniques can be proposed such as voting systems where the results of several techniques could provide better results than using each of them independently. Another interesting open research question is related to the impact in extractive SPL adoption of the results obtained with feature location techniques. We need more empirical analysis of what is the actual meaning of precision and recall by measuring the time and effort required by domain experts to fully locate the features after applying these techniques (i.e., manually removing false positives and adding false negatives).

References

[1] M. P. Robillard, Automatic generation of suggestions for program investigation, in: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005, ACM, 2005, pp. 11–20. doi:10.1145/1081706.1081711.

[2] M. P. Robillard, G. C. Murphy, Concern graphs: finding and describing concerns using structural program dependencies, in: Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA, ACM, 2002, pp. 406–416. doi:10.1145/581339.581390.

[3] C. W. Krueger, Easing the transition to software mass customization, in: Software Product-Family Engineering, 4th International Workshop, PFE 2001, Bilbao, Spain, October 3-5, 2001, Revised Papers, Vol. 2290 of

Lecture Notes in Computer Science, Springer, 2001, pp. 282–293. doi:10.1007/3-540-47833-7_25.
URL http://dx.doi.org/10.1007/3-540-47833-7_25

[4] L. M. Northrop, P. C. Clements, et al., A Framework for Software Product Line Practice, Version 5.0, www.sei.cmu.edu/productlines/framework.html (2009).

[5] K. Pohl, G. Böckle, F. Van Der Linden, Software product line engineering: foundations, principles, and techniques, Springer, 2005.

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

[7] J. Rubin, M. Chechik, A survey of feature location techniques, in: Domain Engineering, Product Lines, Languages, and Conceptual Models, Springer, 2013, pp. 29–58. doi:10.1007/978-3-642-36654-3_2.

[8] W. K. G. Assunção, S. R. Vergilio, Feature location for software product line migration: a mapping study, in: 18th International Software Product Lines Conference - Companion Volume, SPLC '14, Florence, Italy, September 15-19, 2014, 2014, pp. 52–59.

[9] J. Martinez, T. Ziadi, M. Papadakis, T. F. Bissyandé, J. Klein, Y. L. Traon, Feature location benchmark for software families using Eclipse community releases, in: ICSR, Vol. 9679 of Lecture Notes in Computer Science, Springer, 2016, pp. 267–283.

[10] S. E. Sim, S. M. Easterbrook, R. C. Holt, Using benchmarking to advance research: A challenge to software engineering, in: Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA, 2003, pp. 74–83.

[11] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, Y. L. Traon, Bottom-up adoption of software product lines: a generic and extensible approach, in: Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015, ACM, 2015, pp. 101–110. doi:10.1145/2791060.2791086.

[12] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, Tech. rep., Carnegie-Mellon University Software Engineering Institute (1990).

[13] C. Kästner, S. Apel, M. Kuhlemann, Granularity in software product lines, in: Proc. of the 30th Inter. Conf. on Soft. Eng. (ICSE), 2008, pp. 311–320.

[14] P. Grünbacher, R. Rabiser, D. Dhungana, M. Lehofer, Model-based customization and deployment of eclipse-based tools: Industrial experiences, in: Intern. Conf. on Aut. Sof. Eng. (ASE), 2009, pp. 247–256.

[15] M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, P. Lahire, Extraction and evolution of architectural variability models in plugin-based systems, Software and System Modeling 13 (4) (2014) 1367–1394.

[16] A. Shatnawi, A. Seriai, H. A. Sahraoui, Recovering architectural variability of a family of product variants, CoRR abs/1606.00137.

[17] M. V. Couto, M. T. Valente, E. Figueiredo, Extracting software product lines: A case study using conditional compilation, in: CSMR, 2011.

[18] Z. Xing, Y. Xue, S. Jarzabek, A large scale linux-kernel based benchmark for feature location research, in: Proced. of Intern. Conf. on Soft. Eng., ICSE, 2013, pp. 1311–1314.

[19] J. Rubin, M. Chechik, Locating distinguishing features using diff sets, in: IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012, 2012, pp. 242–245. doi:10.1145/2351676.2351712.

[20] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, A. Egyed, Enhancing clone-and-own with systematic reuse for developing software variants, in: Proceedings of International Conference on Software Maintenance and Evolution (ICSME), 2014, 2014, pp. 391–400.

[21] T. Ziadi, C. Henard, M. Papadakis, M. Ziane, Y. L. Traon, Towards a language-independent approach for reverse-engineering of software product lines, in: Symposium on Applied Computing, SAC 2014, 2014, 2014, pp. 1064–1071.

[22] R. Al-Msie'deen, A. Seriai, M. Huchard, C. Urtado, S. Vauttier, H. E. Salman, Feature location in a collection of software product variants using formal concept analysis, in: Proc. of Intern. Conf. on Soft. Reuse, ICSR 2013, 2013, pp. 302–307.

[23] J. Font, M. Ballarin, O. Haugen, C. Cetina, Automating the variability formalization of a model family by means of common variability language, in: SPLC, 2015, pp. 411–418.

[24] H. E. Salman, A. Seriai, C. Dony, Feature location in a collection of product variants: Combining information retrieval and hierarchical clustering, in: Intern. Conf. on Sof. Eng. and Know. Eng. SEKE, 2014, pp. 426–430.

[25] Y. Xue, Z. Xing, S. Jarzabek, Feature location in a collection of product variants, in: Proc. of Working Conf. on Rev. Eng., WCRE 2012, 2012, pp. 145–154.

[26] H. E. Salman, A. Seriai, C. Dony, Feature-to-code traceability in a collection of software variants: Combining formal concept analysis and information retrieval, in: Intern. Conf. on Inform. Reuse and Integr.IRI, 2013, pp. 209–216.

[27] B. Ganter, R. Wille, Formal Concept Analysis: Mathematical Foundations, 1st Edition, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[28] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, R. A. Harshman, Indexing by latent semantic analysis, JASIS 41 (6) (1990) 391–407. doi:10.1002/(SICI)1097-4571(199009)41:6<391::AID-ASI1>3.0.CO;2-9.

[29] R. E. Lopez-Herrejon, D. S. Batory, A standard problem for evaluating product-line methodologies, in: Generative and Component-Based Software Engineering, Third International Conference, GCSE 2001, Erfurt, Germany, September 9-13, 2001, Proceedings, 2001, pp. 10–24.

[30] R. E. Lopez-Herrejon, J. Ferrer, F. Chicano, E. N. Haslinger, A. Egyed, E. Alba, Towards a benchmark and a comparison framework for combinatorial interaction testing of software product lines, CoRR abs/1401.5367.

[31] D. Benavides, S. Segura, A. R. Cortés, Automated analysis of feature models 20 years later: A literature review, Inf. Syst. 35 (6) (2010) 615–636. doi:10.1016/j.is.2010.01.001.

[32] P. Trinidad, D. Benavides, A. R. Cortés, S. Segura, A. Jimenez, FAMA framework, in: SPLC 2008, Limerick, Ireland, 2008. doi:10.1109/SPLC.2008.50.

[33] S. Segura, J. A. Galindo, D. Benavides, J. A. Parejo, A. R. Cortés, Betty: benchmarking and testing on the automated analysis of feature models, in: Sixth International Workshop on Variability Modelling of Software-Intensive Systems, Leipzig, Germany, January 25-27, 2012. Proceedings, 2012, pp. 63–71.

[34] J. Liu, D. Batory, C. Lengauer, Feature oriented refactoring of legacy applications, in: Proceedings of the 28th International Conference on Software Engineering, ICSE '06, ACM, New York, NY, USA, 2006, pp. 112–121. doi:10.1145/1134285.1134303.

[35] C. Kästner, A. Dreiling, K. Ostermann, Variability mining: Consistent semi-automatic detection of product-line features, IEEE Trans. Software Eng. 40 (1) (2014) 67–82. doi:10.1109/TSE.2013.45.

[36] V. Alves, P. Matos, L. Cole, A. Vasconcelos, P. Borba, G. Ramalho, Extracting and evolving code in product lines with aspect-oriented programming, Trans. Aspect-Oriented Software Development 4 (2007) 117–142. doi:10.1007/978-3-540-77042-8_5.

[37] Y. Yang, X. Peng, W. Zhao, Domain feature model recovery from multiple applications using data access semantics and formal concept analysis, in: WCRE, 2009.

[38] R. Al-Msie'deen, A. Seriai, M. Huchard, C. Urtado, S. Vauttier, H. E. Salman, Mining features from the object-oriented source code of a collection of software variants using formal concept analysis and latent semantic indexing, in: SEKE, 2013.

[39] J. Rubin, M. Chechik, Combining related products into product lines, in: Fundamental Approaches to Software Engineering, FASE 2012, Tallinn, Estonia, 2012, 2012. doi:10.1007/978-3-642-28872-2_20.

[40] D. Méndez-Acuña, J. A. Galindo, B. Combemale, A. Blouin, B. Baudry, G. L. Guernic, Reverse-engineering reusable language modules from legacy domain-specific languages, in: Software Reuse: Bridging with Social-Awareness - 15th International Conference, ICSR 2016, Limassol, Cyprus, June 5-7, 2016, Proceedings, Vol. 9679 of Lecture Notes in Computer Science, Springer, 2016, pp. 368–383. doi:10.1007/978-3-319-35122-3_24.

[41] I. E. Araar, H. Seridi, Software features extraction from object-oriented source code using an overlapping clustering approach, Informatica 40 (2).

[42] L. Linsbauer, F. Angerer, P. Grünbacher, D. Lettner, H. Prähofer, R. E. Lopez-Herrejon, A. Egyed, Recovering feature-to-code mappings in mixed-variability software systems, in: 30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014, IEEE Computer Society, 2014, pp. 426–430. doi:10.1109/ICSME.2014.67.

[43] R. Al-Msie'Deen, Mining Feature Models from the Object-Oriented Source Code of a Collection of Software Product Variants, in: ECOOP: European Conference on Object-Oriented Programming, Montpellier, France, 2013, pp. 1–10.

[44] K. Chen, V. Rajlich, Case study of feature location using dependence graph, after 10 years, in: The 18th IEEE International Conference on Program Comprehension, ICPC 2010, Braga, Minho, Portugal, June 30-July 2, 2010, 2010, pp. 1–3. doi:10.1109/ICPC.2010.40.

[45] I. S. Souza, R. Fiaccone, R. P. de Oliveira, E. S. D. Almeida, On the relationship between features granularity and non-conformities in software product lines: An exploratory study, in: 27th Brazilian Symposium on Software Engineering, SBES 2013, Brasilia, Brazil, October 1-4, 2013, 2013, pp. 147–156. doi:10.1109/SBES.2013.12.

[46] Apache, Opennlp, http://opennlp.apache.org (2010).

[47] M. F. Porter, Snowball: A language for stemming algorithms, http://snowball.tartarus.org (2001).

[48] G. Salton, A. Wong, C. S. Yang, A vector space model for automatic indexing, Commun. ACM 18 (11) (1975) 613–620. doi:10.1145/361219.361220.

[49] P. Jaccard, Étude comparative de la distribution florale dans une portion des Alpes et des Jura, Bulletin del la Société Vaudoise des Sciences Naturelles 37 (1901) 547–579.

[50] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, Y. L. Traon, Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines, IEEE Trans. Software Eng. 40 (7) (2014) 650–670.

[51] C. Henard, M. Papadakis, G. Perrouin, J. Klein, Y. L. Traon, PLEDGE: a product line editor and test generation tool, in: 17th International Software Product Line Conference co-located workshops, SPLC 2013 workshops, Tokyo, Japan - August 26 - 30, 2013, ACM, 2013, pp. 126–129. doi:10.1145/2499777.2499778.

[52] C. Henard, M. Papadakis, M. Harman, Y. Le Traon, Combining multi-objective search and constraint solving for configuring large software product lines, in: Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15, 2015, pp. 517–528.

[53] D. L. Berre, A. Parrain, The sat4j library, release 2.2, JSAT 7 (2-3) (2010) 59–6.