A novel approach for Software Architecture Product Line Engineering

Mohamed Lamine Kerdoudi, Tewfik Ziadi, Chouki Tibermacine, Salah Sadou

A Novel Approach for Software Architecture Product Line Engineering

Mohamed Lamine Kerdoudi[a], Tewfik Ziadi[b], Chouki Tibermacine[c], Salah Sadou[d]

[a]*Computer Science Department University of Biskra, Algeria*
L.Kerdoudi@univ-biskra.dz
[b]*Sorbonne Université CNRS, LIP6, F-75005 Paris, France*
Tewfik.Ziadi@lip6.fr
[c]*LIRMM, CNRS and Montpellier University, France*
Chouki.Tibermacine@lirmm.fr
[d]*IRISA, University of South Brittany, France*
Salah.Sadou@irisa.fr

Abstract

A large software system exists in different forms, as different variants targeting different business needs and users. This kind of systems is provided as a set of "independent" products and not as a "single-whole". Developers use ad-hoc mechanisms to manage variability. We defend a vision of software development where we consider an SPL architecture starting from which the architecture of each variant can be derived before its implementation. Indeed, each derived variant can have its own life. In this paper, we propose a novel approach for Software Architecture Product Line (SAPL) Engineering. It consists of, i) a generic process for recovering an SAPL model which is a product line of "software architectures" from large-sized variants. ii) a forward-engineering process that uses the recovered SAPL to derive new customized software architecture variants. The approach is firstly experimented on thirteen Eclipse variants to create a new SAPL. Then, an intensive evaluation is conducted using an existing benchmark which is also based on Eclipse IDE. Our results showed that we can accurately reconstruct such an SAPL and derive effectively pertinent variants. Our study provides insights that recovering SAPL and then deriving software architectures offers good documentation to understand the software before changing it.

Keywords: Software Architecture; SPLE; Software Architecture Product Line; BUT4Reuse; Software Architecture Recovery; Component/Service-based Software

1. Introduction

Software Product Line Engineering (SPLE) aims to improve reuse by focusing not on the development of a single software product but on a family of related products. The systems in a Software Product Line (SPL) approach are developed from a common set of assets in a prescribed way, in contrast to being developed separately, from scratch, or in an ad-hoc manner. This production economy makes the software product line approach attractive. SPLE considers the existence of a single architecture model describing all the variants that implement different software products of a single product line. The particularity of this "single" architecture model is that it includes what is refereed as *a variability model* (also called *feature model*), in which *variability* and *commonality* are explicitly specified using high level characteristics of the so-called *features* [1]. These are then mapped to components, which are organized according to the identified features. Specific software variants can be *derived* (generated) by choosing from the feature model a set of desired features, then SPL tools choose and assemble the appropriate components mapped to the selected features [1].

During recent years, multiple approaches have been proposed addressing SPL implementation, or software product derivation [1, 2]. However, there are many software systems that exist as several "independent" software variants and not as a "single whole". Indeed, large component software systems exist in different forms, as different software variants targeting different business needs and users. For example, IDEs like Eclipse exist as several variants targeting different kinds of software engineers [3]. These software variants often use ad-hoc mechanisms to manage variability and they do not take complete benefits from the SPLE framework. For developers of new software variants that are built upon existing ones, the presence of a single model describing the architecture of the whole system with an explicit specification of commonality and variability is of great interest [4, 5]. Indeed, this enables to see the common part of the whole, on top of which new functionality can be built, in addition to the different features they can use.

In this work, we defend a vision of software development where we consider an SPL architecture starting from which the software architecture of each software variant can be derived. Indeed, each derived software variant can have its own life. This life is regulated by evolution needs whose origin often depends on the context which is specific to each software. From the point of view of the responsible of the software maintenance, the architecture is a crucial artifact for two reasons [6, 7]: i) understand the software before making changes on it, and ii) notify changes made on the software to keep its documentation compliant with its implementation. However, the situa-

tion where the software variants do not have their own/proper architecture raises problems during the maintenance stage of a software on the two points mentioned above: i) referring to a generic architecture to understand a given software is a very difficult task. Knowing that comprehension is the most costly activity during maintenance [8], this will generate considerable additional costs; ii) modifying a generic architecture, to take into account the modifications made on one of its software products, is a task that is not only difficult and error prone, but also with unforeseeable consequences on the other software products. Our vision is that the different software variants can be created from the same SPL, but must have their standalone software architectures to be able to evolve independently and without constraints. However, it is commonly known that having the software architecture of a system is better than dealing with its source code [9].

Our approach for solving the two problems mentioned above is that the product line must first produce the software architecture of a software product, before its corresponding software artifact. This paper considers the challenge of analyzing the source code and the software architectures of existing variants of component-based software systems to reverse-engineer a software architecture to all the existing software variants. We call this constructed architecture a *Software Architecture Product Line* (SAPL) that represents the unique software architecture that supports the software product line and common to all the software variant members of the SPL.

Most of existing SPL extractive approaches focus only on source code [10, 11]. They mainly recover feature models from the source code and maintain traceability links between each feature and its associated code fragments. In our case, we recover SAPL including a special kind of feature models, where features are related to architecture fragments. In addition, the obtained SAPL enables thereby to derive a software architecture for a given product rather than only showing traceability links. Besides, in the literature, and to our best knowledge, there are few works that combine in a complete process the benefits of software architecture recovery techniques with SPL extractive approaches. Such works were analyzed and discussed in a mapping study [12], where the authors state that it is unclear how software architecture techniques which have been mostly developed for a single system can be utilized effectively in an SPL context.

In this work, we propose a novel approach for Software Architecture Product Line Engineering. The overall process of our approach was initially introduced in our previous work [13], which is substantially extended in this paper according two main dimensions: i) A more detailed and extended specification of the two steps. In particular, we describe the SAPL-Forward Engineering step in a new complete way, and ii) a new larger experimentation. This approach consists of a complete process that aims to exploit the benefits of software architecture recovery techniques for single systems in the context of SPL. The proposed approach is composed of two processes: i) a process for SAPL-reverse-engineering that extends the BUT4Reuse framework, which is considered as one of the most effective methods for SPL-reverse-engineering [14, 15]. This framework was proposed as a generic and extensible framework for SPL reverse-engineering. For enabling extensibility, BUT4Reuse relies on *adapters* for the different artifact types. These adapters are implemented as the main components of the framework. Several adapters covering a wide range of artifact types are already available [16]. In this work, we followed the extensibility mechanisms of the BUT4Reuse Framework to implement a new *adapter* for SAPL reverse-engineering from large component-based software systems from a collection of their existing variants. The produced SAPL architectures are of great interest since they enable to see the variability points in the software variants as well as maintain the dependency between these variants [4, 5]. ii) a forward engineering process that uses the recovered SAPL to derive new customized software architecture variants. Several configurations can be created starting from this SAPL. They represent an exhaustive enumeration of all the possible valid configurations. In this process, the discovered constraints from the bottom-up process are used to derive valid and consistent variants. Thus, we followed the extensibility mechanisms of the FeatureIDE Framework [17] to develop a software architecture *composer* that allows to select starting from the SAPL a set of desired features (a possible configuration) that meet a given set of user requirements and derive the software architecture of the new variant.

The approach is firstly experimented on thirteen Eclipse IDE variants to create a new SAPL. Then, an intensive evaluation is conducted using an existing benchmark which is also based on Eclipse IDE. We built the architecture model of Eclipse IDE SPL and derive new software architecture variants. The results of the experiments showed that our approach can effectively reconstruct such an SAPL and derive valid and pertinent variants. One of the insights that can be provided based on our study is that recovering SAPL is of great interest since it allows to derive the software architectures of new variants before their implementations. This is an important activity in software maintenance and evolution since it offers good documentation to understand the software product before changing it.

The remaining of the paper is organized as follows. In Section 2, we expose background material about Software Product Line Engineering and the extractive adoption of SPLs. We also introduce an example which serves as a running example for illustrating our proposals. Section 3 presents a general picture of the proposed approach. In Section 4, we expose our SAPL-Reverse Engineering process, the proposed SAPL Metamodel for Component-Based Software Variants, and its instantiation for the OSGi systems. Section 5 describes our SAPL-Forward Engineering Process. We show the results of our experiments in Section 6. We finally discuss the related work in Section 7, before concluding the paper in Section 8.

2. Background & Problem Illustration

Many development settings of software systems start from a software architecture, which is particularly necessary for large-scale systems. Reusing software architectures across a set of related systems allows to maximize the return on investment of

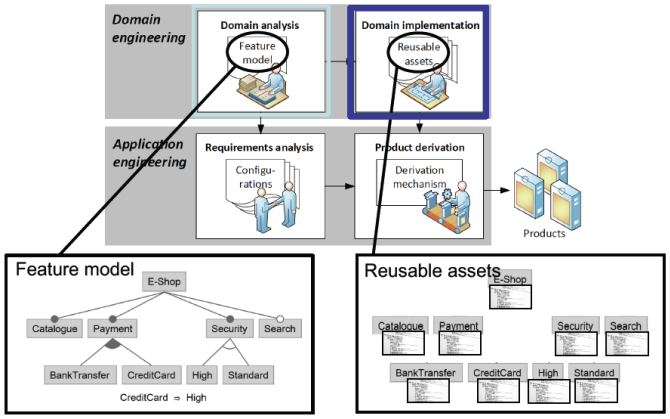

Figure 1: SPLE Process

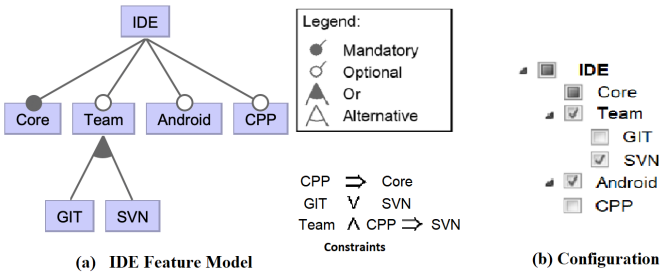

Figure 2: SPL Feature Model

time and effort. Indeed, we leverage the good practices (patterns, styles, etc.) and thereby the quality attributes implemented in this architecture. There are many ways this happens in practice. Indeed, several systems or products resemble each other more than they differ. This is an opportunity for reusing the architecture across these similar products. Thus, SPL simplify the creation of new members of a family of similar systems. We present in this section relevant concepts related to Software Product Line Engineering.

2.1. Software Product Line Engineering

The Software Engineering Institute at Carnegie Mellon University defines a Software Product Line (SPL) as a set of systems sharing 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 [18].

The Software Product Line Engineering paradigm separates two processes that are illustrated in Figure 1 [19]: *i) Domain engineering*: this process is responsible for establishing the reusable software artifacts (assets) such as requirements, design, realisation, tests, etc. and thus for defining the commonality and the variability of the product line. Traceability links between these artifacts facilitate systematic and consistent reuse. *ii) Application engineering*: this process is responsible for deriving product line applications from the software artifacts established in domain engineering. It exploits the variability of the product line and ensures the correct binding of the variability according to the applications' specific needs.

Features of the SPL are specified in what is called a variability model (a.k.a. *feature model*). Feature models (FM) are widely used in SPLE to describe both variability and commonality in a family of product variants [20]. The graphical representation of a feature model is a tree where each feature has a parent feature except for the root feature. Each feature is decomposed into one or more features. In order to derive a new product variant, we need to select a set of features that meet the rules (mandatory, optional, or, alternative) given by the feature model. The selection of a feature implies that its parent is also included. Figure 2 shows an example of a feature model regarding an illustrative and simplified example of IDE architecture. The `IDE` FM consists of a mandatory feature `Core`, two possible `Team` functionalities from which one or both could be selected, two optional `Android` and `CPP` features. The concept of core assets refers to the software artifact needed to implement the SPL.

Furthermore, cross-tree constraints can be specified to define further relationships between features (not in parental relationship). These constraints are arbitrary propositional formulas which must be valued to true. Adding constraints between features can provide more reliable definition of the variability model [21]. Two kinds of cross-tree constraints can be used for any pair of features, namely *requires* and *excludes* constraints. For instance, in Figure 2, the following constraints are:

- "CPP" *requires* "Core": which means, if CPP is included then Core must also be included.
- "GIT" *excludes* "SVN": which means, if GIT is included then SVN should not be included, and vice versa.

2.2. Extractive Adoption of SPLs

Besides, SPL reverse-engineering approaches consider as input a set of existing variants and propose a solution to construct the SPL. This mainly includes the identification of the features, the synthesis of the feature model and the extraction of the reusable assets [14]. Many SPL extraction approaches have been proposed in the last years. Assunção et al. [11, 22] present a complete survey and a systematic mapping on these existing works. Among them the BUT4Reuse framework that we use in this paper.

BUT4Reuse [14, 23] framework is considered as one of the most popular Frameworks that provides a unified environment for mining software artifact variants. It is a generic and extensible framework for extractive SPL adoption. It is generic because it can be used in different scenarios with product variants of different software artifact types (e.g., source code in Java, C, models, requirements, or plugin-based architectures). It is extensible by allowing to add different concrete techniques or algorithms for the relevant activities of extractive SPL adoption (i.e., feature identification, feature location, mining feature constraints, extraction of reusable assets, feature model synthesis and visualizations). Several validation studies of BUT4Reuse using different software artifact types or different extensions have already been published [21, 24, 25].

2.3. Problem Illustration

For illustrating the problem, we use the systems that are developed under the OSGi framework such as Eclipse IDE. The OSGi specification defines a component model and a framework for creating highly modular Java systems [26]. The architecture of Eclipse is fully developed around the notion of *plugin* conforming with the OSGi standard. Eclipse-based IDEs run on top of Equinox[1] which is the reference implementation of the OSGi specification. These IDEs are a collection of similar software products that share a set of software assets. The Eclipse Foundation provides integrated development environments (IDEs) targeting a variety of developers. It offers a set of "software products" (following Eclipse terminology they are called "packages") where each one is a large-sized system composed of hundreds to thousands of components, registering and consuming hundreds of services. This complex structure requires a considerable effort to understand all dependencies when building a new Eclipse IDE software variant.

Currently, if a developer wants to create a customized Eclipse-based IDE, she/he has to select one of the default products[2] (for instance, IDE for C/C++ Developers) and then manually install new features which meet her/his requirements, before adding new functionality to the IDE. Besides, for a given set of Eclipse IDE variants, it is not easy to see the variability points among them. Developers often use ad-hoc mechanisms to manage variability and they do not benefit from the SPLE framework. It is difficult to create a new customized Eclipse IDE variant that only contains a set of desired features (not all the predefined features of an existing product). In fact, the developers should manually analyze and understand the components of the Eclipse IDE variants to identify the common features and then adding the desired features. This task is a cumbersome and error-prone activity for a developer especially that in most cases Eclipse IDEs are too large and complex. In addition, the different Eclipse IDE variants that can be derived from the same SPL, must be able to evolve independently and without constraints.

In this paper, we consider an SAPL as a model starting from which the new customized software product variants can be derived. We aim to adopt the SAPL approach in order to be able to develop efficiently a new customized Eclipse IDE (Software architecture and its implementation). Thus, this SAPL maintains the dependency between the different variants and makes it possible to have specific documentation for each of the software variants and therefore to be able to maintain and evolve independently. In the following sections, we use Eclipse-based IDEs to illustrate our solutions, but the proposed approach is generic and is not related to OSGi or Eclipse.

3. Approach Overview

In this section, we provide an overview of our solution which consists of a novel approach for software architecture product line engineering. Indeed, most of the existing extractive approaches in the literature focus on the feature model extraction from the source code of a collection of software variants. As we aforementioned in the introduction, there are few works that combine the benefits of software architecture recovery techniques with SPL extractive approaches. In our approach, we reverse-engineer the SPL source code in order to extract the SAPL where commonality and variability between fragment of architectures are explicitly specified. The recovered SAPL includes a special kind of feature models, where features are related to architecture fragments. The produced SAPL is used then to derive new software architecture variants. These architectures are important for the maintenance and evolution needs. Thus, in this paper, we propose to revisit the SPL problem from the software architecture (SA) perspective.

In this context, we identified five main challenges: i) How to extract a software architecture from the source code of each variant; ii) How to compare the software architecture variants to identify the common parts and find then different features; iii) How to construct the SAPL with an explicit specification of the variability at an architectural level; iv) How to simplify and reduce the complexity of the recovered architectures. The extraction should be generic and extensible to support all these different aspects; v) Once the SAPL is constructed, one remaining challenge is related to the derivation of new variants. How the SAPL can be used to derive new pertinent SA variants?

This paper proposes an approach to cover all these challenges. Our approach consists on a complete process that aims to exploit the benefits of software architecture recovery techniques for single systems in the context of SPL. It proceeds first by analyzing the source code of existing software variants to extract the software architecture of each variant. The source code of these software variants is created using opportunistic reuse (extractive adoption of SPLs). Our approach supports also the reconstruction of the architectures from products that already belong to an SPL. After that, we reverse-engineer a software architecture called SAPL following that is common to all these software architecture variants. This SAPL is built with an explicit specification of commonality and variability. Second, this SAPL can be used in a SPLE's derivation process in order to derive new customized variants (software architectures and their implementation). The developer is involved to select which features that represent a possible configuration for generating a given variant. The overall process of our approach is illustrated in Figure 3. It is composed of two main sub-processes (in Figure 3): i) A Bottom-Up Process for Recovering SAPL; this sub-process starts first with the Reverse-Engineering of Software Architectures from the source code of each software variant (we call this: "step 0"). Second, it reconstructs an SAPL for these software architecture variants and ii) A SAPL Forward-Engineering Process which allows to derive new variants (Software Architecture Variants).

In the next sections, we describe in detail each sub-process.

[1] `https://www.eclipse.org/equinox/`

[2] available here: `https://www.eclipse.org/downloads/packages/release`

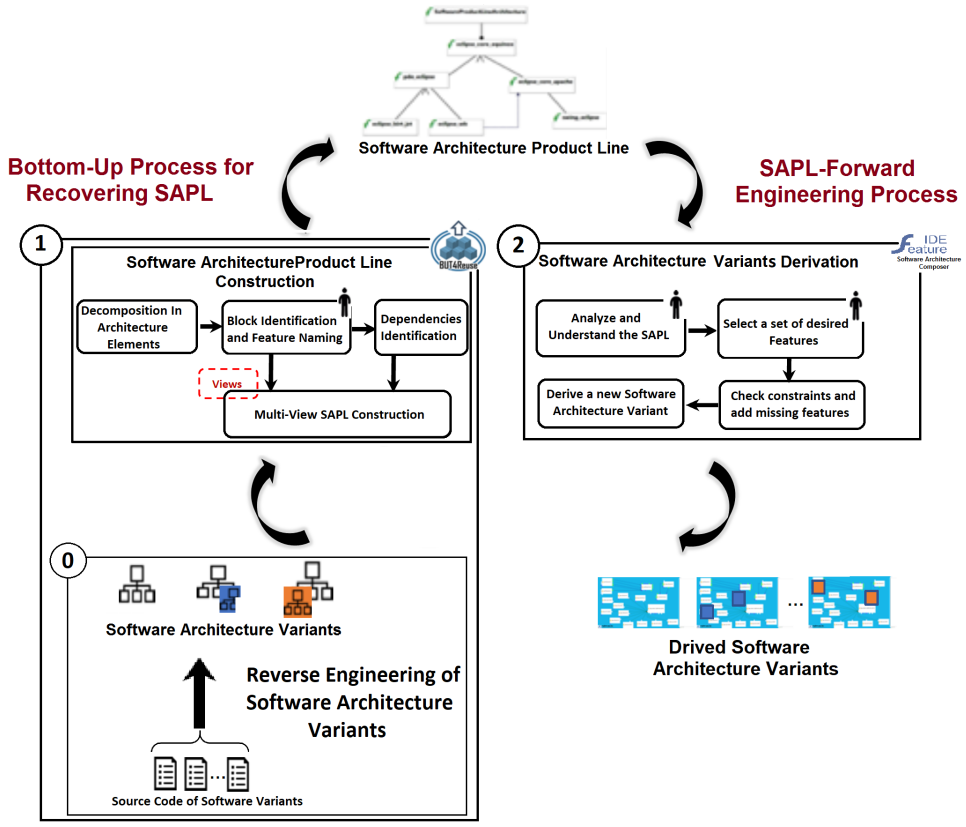

Figure 3: Proposed Approach for SAPL Engineering

4. Bottom-Up Process for Recovering SAPL

Before presenting each step, we first describe the generic meta-model that is supported by our approach.

4.1. SAPL Meta-model for Component-Based Software Architecture Variants

Figure 4 depicts our generic SAPL meta-model which is used for creating an architecture for a set of component-based software variants. To specify the variability model, we have been inspired in the definition of this meta-model by the feature meta-model in [27]. As mentioned above, a feature model is defined with a set of features that can be related by constraints and operators such as *alternative*, *choice*, *optional* and *xor*. So, the left part of the meta-model of Figure 4 shows the introduced concepts to specify feature models. We enriched it by adding component-based architecture elements. An instance of this meta-model serves as a feature model that represents the variability in a family of software product variants and a comprehensive architecture (modules / components) that helps the developer to understand the structure of the SPL features and the relations between them.

As our meta-model is used for representing component-based systems, it has been defined based on top of an abstract syntax of a software component model. The latter is used to represent any kind of component-based system such as an OSGI or a Spring-based one. A generally accepted definition of a software

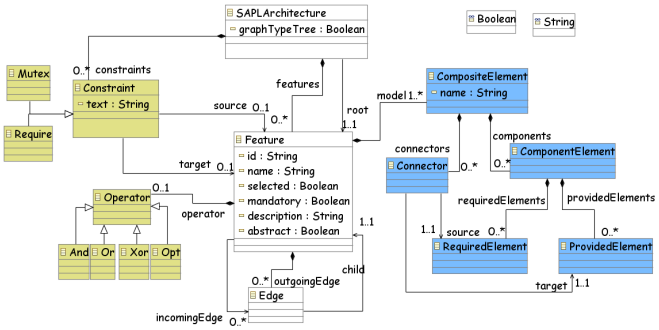

Figure 4: SAPL Metamodel for Component-Based Software Architecture Variants

component is that it is a software unit with provided capabilities and a set of requirements. The provided capabilities (*ProvidedElement* in our meta-model) can include operations the component is able to execute. The requirements (*RequiredElement* in our meta-model) are needed by the component to produce the provided capabilities.

4.2. Mapping of the SAPL Metamodel to OSGi Component Model

We show in this sub-section how to instantiate our generic SAPL meta-model (in Figure 4) for a concrete component based system which is related to the OSGi System. Figure 5 presents the result of the instantiation for OSGi component model. In-

deed, a component in OSGi is known as a bundle or a plugin (*PluginElement* in this meta-model) which packages a set of Java types, resources and a manifest file. Plugin dependencies are expressed as manifest headers that declare requirements and capabilities. The "import-package" header is used to express a plugin's dependency upon packages that are exported by other plugins. The "require-bundle" is used when a plugin requires another plugin. The first plugin has access to all the exported packages of the second. The manifest file declares also what are the packages that are externally visible using "export-package" (the remaining packages are all encapsulated). Furthermore, the Java interfaces that are present in the exported and imported packages are considered respectively as the plugin's provided and required interfaces (represented by *ProvidedInterfaceElement* and *RequiredInterfaceElement*).

Besides, the OSGi framework introduces a service-oriented programming model which is a publish, find and bind model. The registered services with the OSGi Service Registry are represented by the *RegisteredServiceElement*, while a consumed service by a plugin is represented by a *ConsumedServiceElement*.

Services are not the only collaboration way between plugins. Equinox provides a means of facilitating inter-plugin collaboration via *Extension Registry*. Plugins open themselves for extension or configuration by declaring extension points (*ExtensionPointElement* in this meta-model) and defining contracts. Other plugins contribute by developing extensions (*ExtensionElement* in this meta-model) using existing extension points.

Our OSGi model allows to produce several software architecture with several points of view that represent different kinds of plug-in's capabilities and requirements. The supported architecture points of view in our model are: *interface*, *service*, *package*, and *extension*. Of course these points of view are not orthogonal, there are intersections between each other. But, we are convinced that the developers would not be able to understand the whole software variant by analyzing all the points of view together. Thanks to this meta-model, developers can progressively understand the software variant by analyzing each architecture view separately. In addition, our framework can be easily extended to support other points of view in order to cover all the aspects that the developers need to know when they develop a new variant.

4.3. Reverse-Engineering of Software Architecture Variants

The first step in our bottom-up process (step 0) uses reverse-engineering techniques to extract a software architecture variant from the source code of each software variants. For instance, the reverse-engineering of software architectures from Eclipse IDE variants is based on the analysis of the configuration files and the source code of the different components (plugins).

Indeed, for recovering the SA variants, we analyze the Eclipse artifacts as follows: i) for each Eclipse variant, we generate a software architecture where the root element is a *compositeElement* with the name of this variant (for instance "Eclipse for Java developers"). ii) for each plug-in in the Eclipse variant, we create a *PluginElement* with the plug-in's characteristics. iii) we parse the manifest file of each plug-in to identify the exported and imported package elements. iv) the provided and required interface elements are identified by analyzing the Java source code and Bytecode (in case source code is not available) in the exported and imported package folders. iv) the extension and extension-point elements are identified by parsing the "*plugin.xml*" files of each plug-in. v) finally, the programmatically registered and consumed services are identified by parsing the source code and bytecode of each class in the plug-in. We parse here the following statements: `<context>.registerService(..)` and `<context>.getServiceReference(..)` to capture the type of classes that are instantiated and registered. In addition, the services that are declared with the DS (Declarative Services) framework are identified by parsing the "*OSGI-INF/component.xml*" files. Before saving the architecture, we create the connectors to link the created elements. The parsing of the source code and bytecode has been implemented by using Java libraries such as AST-Parser for source code parsing and ObjectWeb's ASM [3] for bytecode parsing.

4.4. SAPL Construction

The software architectures variant that are recovered in the previous step from the source code of the software variants are used as input for SAPL construction step. Thus, the different software architecture variants are analyzed and compared to identify the common part and the different features. As illustrated in Figure 3, this activity extends the BUT4Reuse framework to support architectural artifacts.

To support the different types of artifacts, and enabling extensibility, BUT4Reuse relies on *adapters* for the different artifact types. These adapters are implemented as the main components of the framework. An adapter is responsible for decomposing each artifact type into its constituting elements, and for defining how a set of elements should be constructed to create a reusable asset. Designing an adapter for a given artifact type requires three main tasks:

- **Element identification**. The first step is to identify the *Elements* that compose an artifact. This will define the granularity of the elements in a given artifact type. For the same artifact type, we can select elements at different levels of granularity (e.g., package level versus statement level for source code).
- **Similarity metrics definition**. This task defines a similarity metric between any pair of Elements. An element should be able to compare its definition with the one of another element and return as output a value ranging from zero (completely different) to one (identical).
- **Structural dependencies definition**. The purpose of this task is to identify *Structural Dependencies* for the *Elements*. When the artifact type is structured, the elements

[3] website : `https://asm.ow2.io/`

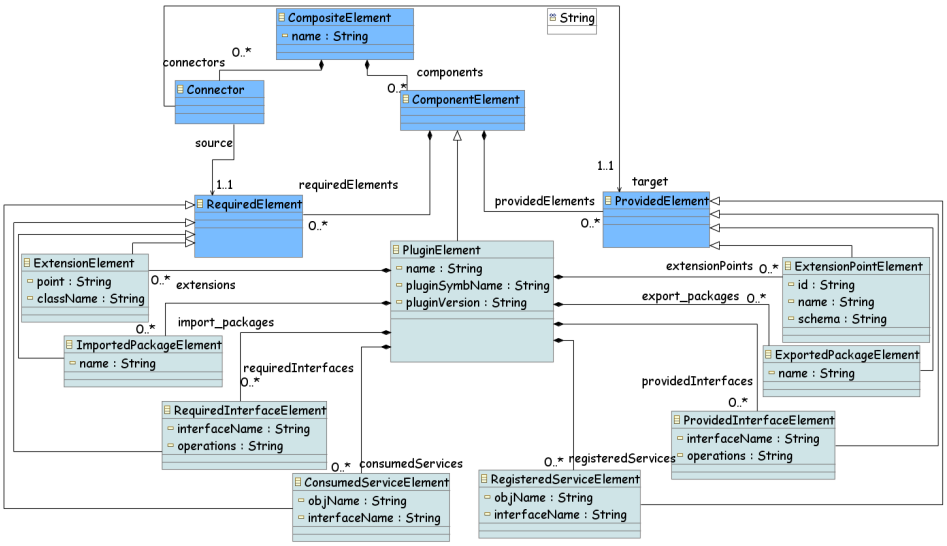

Figure 5: Modeling of OSGi Elements

will have containment relations. In the case of architecture artifacts, relations between interfaces, components and plugins usually capture this information.

In this paper, we extend BUT4Reuse by proposing a new adapter related to Eclipse-Software Architectures [4]. In addition to allow comparing software architectures, this new adapter is designed with a set of parameters to consider different architectural points of view (services, interfaces, packages and extensions).

Once the adapter is implemented, SAPL construction follows four sub-activities as illustrated in Figure 3.

Decomposition in Architectural Elements. The first step takes as input a collection of architecture variants that are obtained from the reverse-engineering activity. It decomposes each variant into a set of Architectural Elements (AEs). The computed AEs can be of different types depending on the considered point of view. For instance, to compare and analyze several Eclipse software variants, BUT4Reuse divides each variant into the following elements: `PluginElement`, `ServiceElement`, `PackageElement`, `ExtensionPointElement`, `ExtensionElement`, and `InterfaceElement`.

To identify them, our adapter loads and parses the input Eclipse SA variants and performs a mapping of the elements in the input SAs with these elements.

Block Identification and Feature Naming. This step reuses algorithms implemented in BUT4Reuse which automatically identify sets of AEs that correspond to the distinguishable features from the SA variants. These sets of AEs are named *Blocks*. *Blocks* permit to increase the granularity of the analysis by the domain experts in order to not reason at Element level. In fact, Block identification represents an initial step before reasoning at feature level.

[4] Available online:`https://github.com/kerdoudi/but4reuse`

In this paper, we reused the algorithm called *Interdependent Elements* that formalizes Block identification using class equivalences [28]. This algorithm is based on a formal definition of a Block that uses the notion of interdependent Elements, which is defined as follows: Given a set Software Architecture Variants (`SAV`), two Architectural Elements e_1 and e_2 (of software architectures from `SAV`) are interdependent if and only if they belong to exactly the same variants of `SAV`. This is defined formally as follows:

- $\exists sav \in \texttt{SAV}\ e_1 \in sav \wedge e_2 \in sav$
- $\forall sav \in \texttt{SAV}\ e_1 \in sav \Leftrightarrow e_2 \in v$

Since interdependence is an equivalence relation on the set of Elements of `SAV` this leads to the following definition of `Block candidates`:

Given `SAV` a set of software architecture variants, a `Block` of `SAV` is an equivalence class of the interdependence relation of Architectural Elements of `SAV`.

In Figure 6, we illustrate an example of using the Blocks identification algorithm. The ellipses represent the software architecture variants. The stars represent Architectural Elements within these artifacts. The similarity metric between Elements establishes when Elements from different artifacts are equal and therefore we can compute the intersections among them. Hence, the separated intersections represent the identified Blocks. For instance, the `Block 0` contains the Elements that are common to all the SA variants, `Block 1` groups elements that are shared only by variant 3 and variant 4.

Once blocks are identified, the next step is a semi-automatic process where domain experts manually review the elements from the identified blocks to map them with the functionalities (i.e., features) of the software variant. BUT4Reuse integrates what is called *VariCloud* [29], a tool that analyzes the elements inside each block and extracts words that help domain experts to identify features. *VariCloud* uses information retrieval techniques, such as TF-IDF (Term Frequency-Inverse

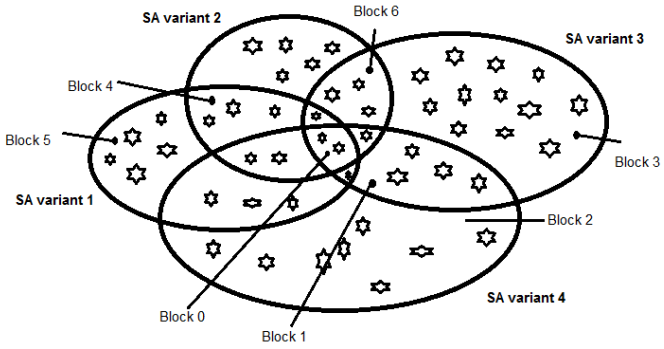

Figure 6: Block Identification from a set of Software Architecture Variants

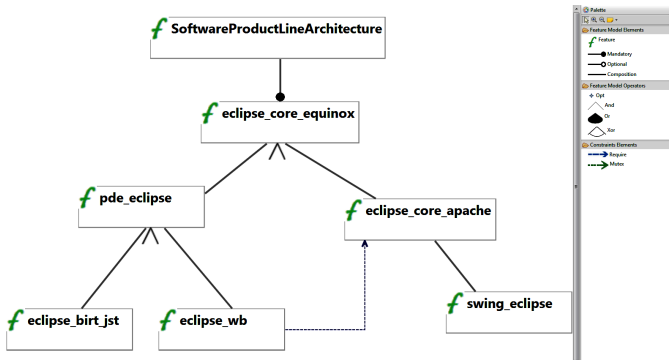

Figure 7: Example of SAPL for three Eclipse Variants (Extension and Package point of view)

Document Frequency), to analyze the text describing elements inside blocks. The descriptions used by BUT4Reuse to build word clouds are thus provided by the specific adapter. As for our adapter, the words correspond to the names of packages, extensions, services, interfaces and plugins. Using these words allows to give automatically more representative names.

Dependencies Identification. During this step, the approach identifies the dependencies between the different blocks. BUT4Reuse uses the dependencies defined within the adapter to identify dependencies between blocks. In our adapter, we extract the *requires* and the *mutual-exclusion* dependencies between blocks based on the element dependencies. This is performed as follow:

- let B1 and B2 are two identified blocks
- "B1 *requires* B2" iff $\exists e_1 \in$ B1$\wedge\exists e_2 \in$ B2$\wedge$ e_1 *requires* e_2;
- "B1 is in *mutual-exclusion dependency* with B2" iff $\exists e_1 \in$ B1 $\wedge$ $\exists e_2 \in$ B2$\wedge$ e_1 is in *mutual-exclusion dependency* with e_2;

Multi-View SAPL Construction. A software architecture of a large system is a complex entity; it cannot be presented in a single point of view. In this step of our process, we enable the developer to construct a multi-view SAPL. These points of view can help and assist the developer to understand progressively the SPL. However, we should not confuse these points of view with architectural views which allow addressing separately the concerns of the various "stakeholders" of the architecture. For instance, the four views: logical view, process view, physical view, development view that have been proposed by Kruchten in [30]. Our points of view are related to abstraction aspects. Developers that want to use our approach are then free to define their own points of view for their components. For example, we have defined for the OSGi component model the following "points of view": service, interface, extension, package.

Now, in the context of Eclipse Software Architectures, we argue that the developer wants first to analyze the extension point of view. The latter gives less complex (in terms of number of elements and connections) SAPL model that can help her/him to understand easily what are the components that are extended and what are the extension-points that are provided to be extended. After that, the interface and service points of view can provide which interfaces or services that should be implemented or consumed.

Besides, the current implementation of the SAPL construction is realized with two distinct algorithms that provide two different organization of features. They are inspired from feature model synthesis in BUT4Reuse [14]. In the first one the features are organized in the SAPL as a Flat feature diagram with all the constraints included as cross-tree constraints. The second one is a heuristic called "Alternatives before Hierarchy" that is based on calculating first the Alternative constructions from the mutual exclusion constraints, and then create the hierarchy using the requires constraints. The constraints that were not included in the hierarchy are added as cross-tree constraints. Moreover, the generation of the SAPL is implemented as a separate plug-in that provides an extension-point for other developers to extend this activity for generating SAPL by using more sophisticated algorithms.

Once the SAPL is recovered, it can be visualized and updated graphically using our SAPL graphical modeling tool that is provided as a set of plugins which are implemented based on the Eclipse Modeling Framework and the Graphical Modeling Framework. Our graphical tool allows also to visualize each feature in the SAPL as a separate fragment of software architecture. Indeed, the tool provides an editor that allows to visualize graphically the SAPL as follows. First, the SAPL can be visualized as a compact representation of all the assets of the SAPL in terms of "features". Second, we enable the developer to click twice on a given feature in order to visualize its architecture, which can be opened in another editor. In this way, instead of visualizing the whole SAPL in one screen, we assist the developer to understand features progressively. For instance, Figure 7 depicts the generated SAPL starting from three Eclipse variants which are IDE for Java, IDE RCP and RAP, and IDE for Java and Report Developers. The feature "`eclipse core equinox`" is common to the three variants. The edge with a dashed line represents a discovered require dependency.

Besides, Figure 8 shows an excerpt of the architecture fragment that represents the feature "`eclipse birt jst`". As we can see, the component "`BIRT Emitter Conf. Plug-in`"

provides an extension-point which is extended by several plug-ins. In this architecture, the set of provided / required elements that are not connected to other components, represent elements that are connected to components located in other features. Our tool enables the developers to merge two or several model fragments (that represent two or several features) in a single architecture fragment which allows to visualize the structural dependencies between these features.

In the current implementation the reconstructed SAPL is related to OSGi Component Based software variants. This plug-in provides an extension-point for other developers to contribute by developing extensions for generating SAPL for other kinds of component-based software product line, such as applications built with Java 9+ module system.

5. SAPL Forward-Engineering Process

The goal of this process is to use the recovered SAPL to create valid configurations and to derive in an effective way new `customized` software variants. SAPL is used for modeling all the possible configurations of the software architecture of a specific domain. It captures the commonalities and the variabilities among these software architectures. In this way, in order to complete the "loop", our forward-engineering process allows the developer to analyze and understood the SAPL that is recovered using the previous bottom-up process and then, derive the new software variant (software architecture and its implementation) as a new variant. As depicted in Figure 3, our software architecture derivation process is composed of the following steps:

1. Analyse and understand the SAPL,
2. Select a set of candidate features,
3. Check the consistency and add missing features,
4. Derive the corresponding variant.

The first step is considered as one of the most important activity in the derivation process. As we know the architecture of a software system abstracts its complex structure as more manageable and comprehensible high-level structure. Thus, the SAPL lets the developer to know the structure of each identified feature and its relationship with the other features.

After understanding the SAPL and its features, the second step in this process consists on selecting starting from this SAPL a set of features that meet the developer's requirements. At the end, before merging them, a consistency check of the selected features is performed based on the discovered constraints.

In fact, features are related to fragments of software architecture that represent a characteristic or a functionality of the software. They can be optional or mandatory. A selection of a number of these features defines one specific configuration of the software architecture. In our derivation process we use the constructed assets (fragments of software architectures) obtained from the bottom-up process as reusable assets. The process is based on merging the selected features. This is defined formally as:

- Let F_{all} be the set of all the features in the recovered SAPL,
- Let F_c be a set of candidate features that are selected by the developer, where,
  - $F_c \subseteq F_{all}$,
  - $F_c = F_m \cup F_o$, with, F_m is the set of mandatory features and F_o is the set of optional features.
- Let F_s be the final set of selected features by the developer, after checking their consistency and adding the missing features, where,
  - $F_s = F_m \cup F'_o \cup F_a$, with,
  - $F'_o \subseteq F_o$, where, $\forall\ f_{op} \in F'_o\ \forall\ f_m \in F_m$, f_{op} is not in *mutual-exclusion dependency* with f_m
  - F_a is the set of missing features that are added after checking the discovered constraints. This is defined formally as follow:
    $((\forall\ f_a \in F_a,\ \exists\ f_m \in F_m,\ \forall f_{m2} \in F_m$, where, "$f_m$ `requires` f_a" $\wedge$ f_a is not in `mutual-exclusion dependency` with f_{m2})
    $\vee\ (\forall\ f_a \in F_a,\ \exists\ f_{op} \in F'_o,\ \forall f_{op2} \in F'_o$, where, "$f_{op}$ `requires` f_a" $\wedge$ f_a is not in `mutual-exclusion dependency` with f_{op2}))

Once the final set of selected features (F_s) is created, we proceed to merging them one-by-one until constructing the new software architecture variant. Merging two features consists on creating a software architecture by applying these steps: i) remove duplicate components from the two features and add the remaining component elements into the architecture. ii) add to each of the created component element their required and provided architecture elements. iii) create connectors that connect the required elements to the provided elements that have the same name and the same point of view (service, interface, extension, etc.).

However, we implemented our software architecture derivation process as a `FeatureIDE composer` that is called a "`software architecture composer`"[5]. It allows to select one possible configuration and to check if it satisfies all the constraints and than derives the corresponding software architecture. The FeatureIDE Framework [17] is an Eclipse-based IDE that supports all phases of feature-oriented software development for the development of software product line: domain analysis, domain implementation, requirements analysis, and software generation. The Feature IDE tool provides extensibility for including `composers` dealing with different artifact types. That means that any DSL can be enriched with variants derivation functionalities.

In order to derive a new Eclipse Software Architecture variant, the developer can use our `FeatureIDE SA Composer`. to

[5]Available online:`https://github.com/kerdoudi/but4reuse`

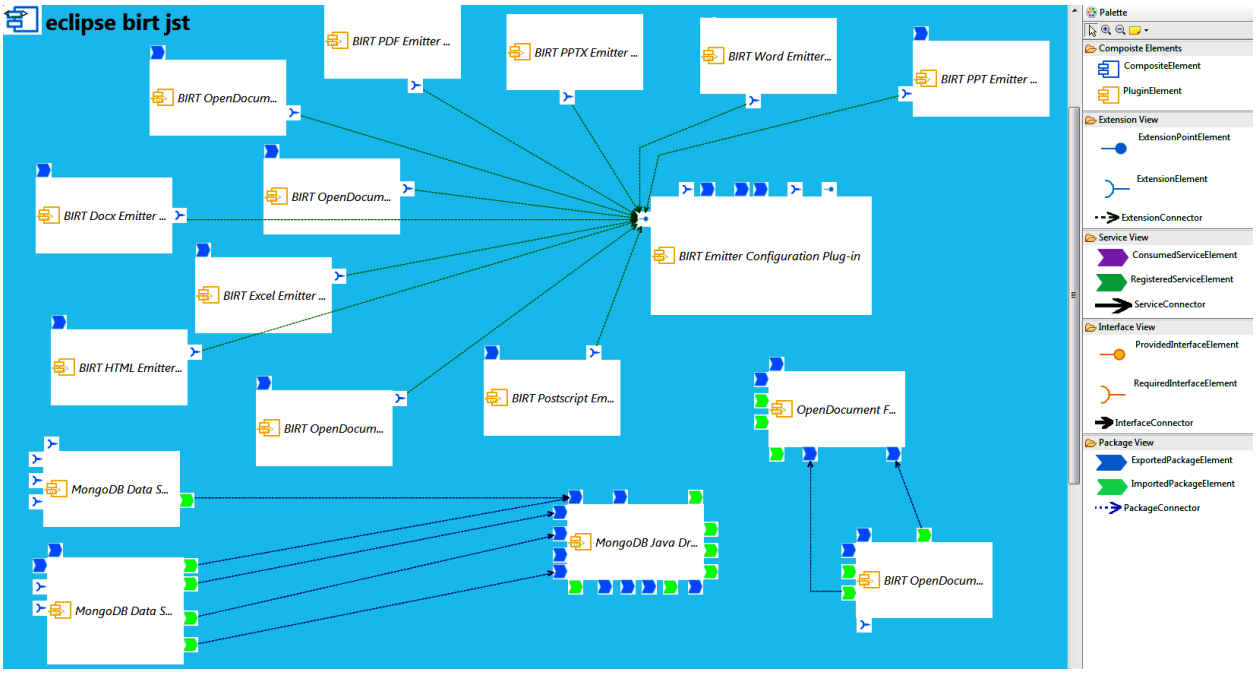

Figure 8: Excerpt of the SA of "`eclipse birt jst`" Feature (Extension and Package point of view)

configure manually the SAPL by selecting a set of desired features among an identified list. After an automatic check of constraints and selecting automatically the features that are meeting constraints, our composer generates a new Software Architecture variant by composing the reusable assets. This architecture model represents the structure of the selected features and their relationships without variability information, which is useful for the understanding purpose. At the end, the new variant is generated by collecting the extracted software assets which correspond to the selected features.

Besides, one of the promising challenges in the SPL context is the continued evolution of the software variants such as when new features are added to the family. Actually, this is related to another problem which is out of the scope of this paper. It is related to co-evolution of product variants and their SPL. In our approach, we must re-execute the complete bottom-up process for recovering the new SAPL. Indeed, theoretically one of the interests of having an SPL is the fact that when bugs occur in the system their correction and evolution are carried out at the SPL level. But, in reality, this is not easy to do. In this respect, the co-evolution of system families is proposed as a novel methodology to deal with these issues (approaches like [31, 32, 33]) and this is not the goal of the presented work.

6. Experimentation and Validation

Our approach includes two processes: a bottom-up process for reconstructing the SPL and a forward-engineering process for deriving new variants from the SPL. In order to evaluate this approach, we conducted a set of experiments to evaluate the two processes.

Thereby, we addressed the following two research questions:

- **RQ1**: What is the performance of our SAPL reconstruction process, in terms of identifying the expected features and artifacts in the product line?

  In the research question RQ1, we measured the performance of the bottom-up process through a controlled experiment. In this experiment, we used a real-world set of Eclipse IDE variants. First, we recovered the SAPL from the set of Eclipse IDE variants. After that, we measured the `precision`, `recall`, and `F1-Score` of the identification of features.

- **RQ2**: Based on the recovered SAPL can we derive in an effective way new customized variants? In other terms, what is the performance of the top-down process for product variant derivation from the SAPL?

  In the research question RQ2, we compared the customized software architecture variants that are derived automatically using our forward-engineering process with the same variants that are created without our approach. We mean by without our approach, choosing a given Eclipse IDE variant and install manually new features (clicking on Help->Install New Software...).

  This part of the experiment is based on the following steps:

  1. We use our `software architecture composer` to compose and derive a set of *new customized variants*. This is performed in this way:
     (a) We created a configuration by selecting from the SAPL the features that are identified from a given input variant (i.e. an Eclipse IDE variant).

(b) After that, we chose from the SAPL one or a set of additional features that are identified from the other input Eclipse IDE variants, and added them to the created configuration.
(c) These additional features should not belong initially to the features of the candidate input variant (selected in step 1a).

2. In the other side, we have taken the candidate input variant (from step 1a), and we have installed manually all the same additional features that are previously chosen.
3. At the end, we used two architectural change metrics to measure the similarity between the derived customized software architectures and the software architectures of the variants that were created manually (in step 2).

6.1. Dataset: Eclipse Variants

In this evaluation, we have used a set of Eclipse IDE variants. We have used in particular two datasets that are considered as input software variants. The first dataset represents a set of official Eclipse IDE variants that contains 13 variants[6] (we have selected the Eclipse 2020-03 R release). The size of these variants varies from 193 to 639 MB and the number of components varies from 383 to 975 components. The second dataset is a set of Eclipse IDE variants that are automatically generated using the EFLBench Framework in [3]. EFLBench Framework integrates an automatic and parametrizable generator of Eclipse IDE variants. It automatically creates variants taking as inputs: i) an Eclipse IDE and ii) the number of variants that we want to generate.

The use of the EFLBench Framework is motivated by the fact that the official Eclipse releases contain few variants (at most 13 variants), which can be considered as a limitation for intensive evaluation of our approach with other scenarios with larger amount of variants. Indeed, regardless of the official versions, in practice, developers create their customized Eclipse IDE by installing/uninstalling projects into an official release. So, in order to achieve the most general results in our experiment we did not limit ourselves to the official Eclipse releases. We decided to use in this experimentation a large number of Eclipse IDE variants. This enables us to empirically analyze whether the number of input variants has an impact on our approach.

6.2. Evaluation Metrics

In this subsection, we explain the metrics that are used in this evaluation for the context of Eclipse IDE variants.

1. **Performance Metrics**
   In order to measure the `precision`, `recall`, and `F1-Score` of our Bottom-Up process, we have compared the content of each identified feature in the recovered SAPL with the content of predefined Eclipse features of the input variants. In fact, in an official Eclipse IDE one or more plugins can be grouped together into an `Eclipse feature` so that a user can easily load, manage, and brand those plugins as a single unit. A predefined `Eclipse feature` describes via a "*feature.xml*" file a list of plugins and other features which can be seen as a logical unit composed of a set of related components. It has a name, a version number and a license information assigned to it. All the predefined `Eclipse features` are located in a specific folder which has the name "*features*" [7].
   In this experimentation, we use the predefined `Eclipse features` of the input variants as a "`ground truth`" to compare them with the identified features using our approach. For instance, in Figure 9 we show an example of what we consider as a ground-truth of three imaginary Eclipse input software variants. In this Figure, for each variant we show its predefined features. For example, "Variant1" contains four features: *f1*, *f2*, *f3*, and *f4*, where each of them groups a set of plugins.
   Now, let us suppose that our approach has been used to recover the SAPL from the three software variants. As a result, a set of features have been identified (if_1, if_2, etc.). In this evaluation, for each identified feature (for instance, if_1), we proceed in this way: we select the "*feature.xml*" files that are located in the common "*features* folders" (the intersection set) of the variants from which this feature is identified. In our example, if we suppose that if_1 has been identified from the three variants, then, we select from the ground-truth in Figure 9 *f1* and *f2* (because they are common to the three variants). After that, we compare the plugins that belong to this identified feature with the plugins that are present in these "*feature.xml*" files (for *f1* and *f2* these plugins are p1, p2, p3, p4, p5, and p6).

   - The **True Positives** (**TP**) are the set of plugins that are assigned to an identified feature and that are correct according to the ground truth (they belong to the predefined Eclipse features). For example, if our approach has assigned to if_1 the following plugins: p1, p2, p3, p4, p5, p7, and p8 then **TP** for if_1 is equal to 5.
   - **False Positives** (**FP**) represent the set of plugins that belong to the candidate feature, and they do not belong to the ground truth. For if_1 **FP** is equal to 2 (p7 and p8).
   - **Precision** represents the ratio of correctly identified plugins to the total number of plugins of a given feature. It is defined by Equation 1.

$$\textbf{Precision} = \frac{TP}{(TP + FP)} \quad (1)$$

   For instance, $\textbf{Precision}(if_1) = \frac{5}{(5+2)} = 0.71$

[6] Downloaded from: `https://www.eclipse.org/downloads/packages/release/2020-03/r`

[7] Predefined features of Dataset 1 are available online: `https://github.com/kerdoudi/Eclipse2020-03RGroundTruthFeatures`

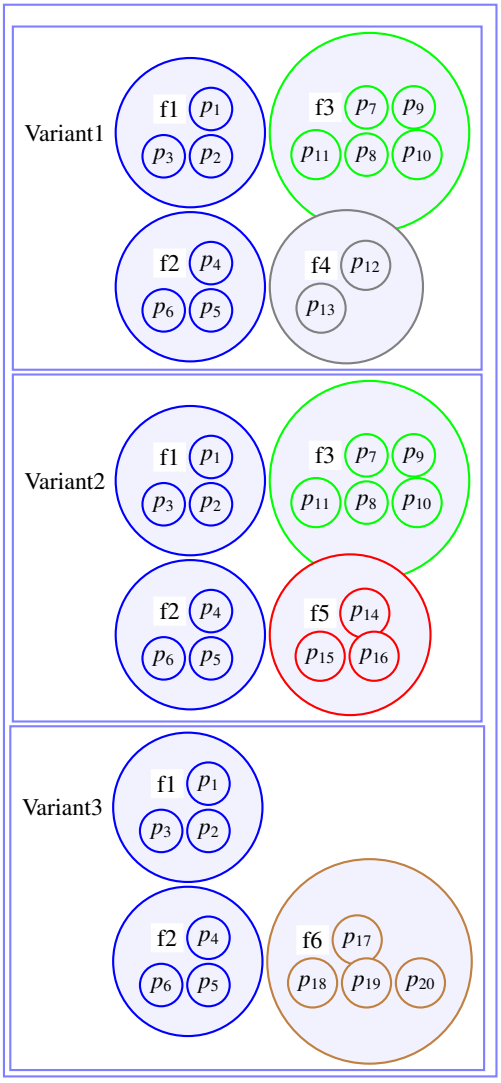

Figure 9: Example of Ground-Truth features of three Imaginary Eclipse Input Software Variants

- According to the ground truth (predefined features), there can be some missing plugins that are not included in the set of the plugins of an identified feature. Those plugins are **False Negatives** (**FN**). For if_1, **FN** is equql to 1. The missed plugin is p6.
- **Recall** is the ratio of correctly identified plugins relative to the plugins that should be identified for the candidate feature. It is defined by Equation 2.

$$\textbf{Recall} = \frac{TP}{(TP + FN)} \tag{2}$$

For instance, $\textbf{Recall}(if_1) = \frac{5}{(5+1)} = 0.83$

High `recall` means that the developers do not have to manually add a lot of missing plugins. Conversely, low recall implies an important involvement of the developers.

A low value for precision means that the identified feature contains plugins that should not belong to the feature. This, also, leads to involvement of the developers in order to manually remove the unwanted plugins.

- We provide also the `F1-Score` measure for evaluating the accuracy. It is a synthetic score of precision and recall measures. It is the weighted average of both (in Equation 3).

$$\textbf{F1-Score} = 2 \times \frac{Precision \times Recall}{(Precision + Recall)} \tag{3}$$

For instance, $\textbf{F1-Score}(if_1) = 2 \times \frac{0.71 \times 0.83}{(0.71+0.83)} = 0.76$

2. **Architectural Change Metrics**

We have used the two metrics: *Architecture2Architecture* (`a2a`) [6] and `MoJoFM` [34] to measure the accuracy of the derived software architectures using our composer. The metrics have been used to measure the architectural change between the derived customized software architectures and the software architectures of the variants that are created manually. The architectural change refers to the addition, removal, and modification of components and their elements (service, interface, extension, etc.). `a2a` is defined by the following formula:

$$a2a(A, B) = (1 - \frac{mto(A, B)}{aco(A) + aco(B)}) \times 100\% \tag{4}$$

where, $mto(A, B)$ is the number of operations needed to transform architecture A into B and $aco(A)$ is the number of operations needed to create architecture A from a "null" architecture. Five operations are allowed to transform one architecture into another: additions ($addE$), removals ($remE$), and moves ($movE$) elements from one architecture to another; as well as additions ($addC$) and removals ($remC$) of components themselves.
`MoJoFM` is defined by the following formula:

$$MoJoFM(A, B) = (1 - \frac{mno(A, B)}{max(mno(\forall A, B))}) \times 100\% \tag{5}$$

where, $mno(A, B)$ is the minimum number of `Move` or `Join` operations needed to transform the architecture A to the architecture B.
A score of 100% indicates that the architecture A is the same as the architecture B. A lower score results in greater disparity between A and B. For example, if an architecture **A** is composed of three components:

- $C1 = \{r_1, r_2, p_1, p_2, p_3\}$
- $C2 = \{r_3, p_4\}$,
- $C3 = \{r_4, r_5, p_5\}$,

and an architecture **B** is composed of two components:

- $C4 = \{r_1, r_2, p_1, p_2, p_3, r_3, p_4\}$
- $C5 = \{r_4, r_5, p_5\}$,

and an architecture **D** is composed of two components:

- $C6 = \{r_1, r_2, p_1, p_2, p_3, r_3\}$
- $C7 = \{r_4, r_5, p_5, p_4\}$

Where, r_i and p_i represent their provided and required elements (interface, extension, services, package).
Thus, the two components $C1$ and $C2$ of architecture **A** can be joined in a simple operation to give $C4$. As a result, **mno(A, B) = 1**. However, **mno(A, D) = 2**, because also the element p_4 must be moved to $C7$.
Thus, the obtained **MoJoFM** scores for these architectures are:

- MoJoFM(A,B) = 87.5%
- MoJoFM(A,D) = 75.0%

This disparity is explained by the fact that MoJoFM's join operation is less expensive, which leads to high scores. As for the a2a scores, we obtained:

- a2a(A,B) = 93.33%
- a2a(A,D) = 93.33%

This is explained by the fact that the candidate architectures have few components which leads to low MoJoFM scores compared to a2a. Actually, the a2a has been designed to address some of MoJoFM drawbacks [6].
In this experimentation, we have used the `LoongFMR` implementation[8] of the two metrics `a2a` and `MoJoFM`. Each program takes as input two architectures given as two RSF files. An RSF file is a representation of an architecture in term of a partition of the architecture elements. Thus, the two metrics are used to calculate the distance between two architectures. We first convert the candidate architectures to RSF files as follows:

- The relation name in the RSF file must always be *"contain"*. Read the following syntax:
  contain componentName objectName
- We must have exactly one line per cluster (component), i.e. only flat decompositions are supported.
- The objects represent the provided and required elements (interface, extension, service and package). We added the terms *Provided* or *Required* before the name of each element to distinguish between them.

The following lines represent an example of an RSF file:
contain com.ibm.icu Providedcom.ibm.icu.text.SCSU
contain com.jcraft.jsch Requiredcom.jcraft.jsch.Cipher
contain com.jcraft.jsch Requiredcom.jcraft.jsch.DH

6.3. Answering Research Question 1

6.3.1. Results of using the first Dataset

First, we have run our adapter for the first dataset (the 13 official Eclipse variants). We have identified 76 features. Figure 10 presents the identified features. It illustrates for each input variant what are the identified features. The common part between all the variants represents the "`Feature 0`" that is named "`Eclipse core equinox`". This represents the core components that must exist in each variant. During the execution, the developers can see the name of any feature just by glassing its cursor on this feature on the left side of Figure 10. It is also possible to automatically generate a table that show a clear distribution of all the identified features over the input variants. In Table 1, we show an excerpt of how features are distributed over the input Eclipse IDE variants.

Moreover, all features in Figure 10 have been assigned automatically to blocks, thanks to the word cloud that is used to name the identified blocks starting from words that are extracted from the elements names (extension, service, interface, etc.). Figure 11 shows the result of automatic feature naming of two identified Blocks. In fact, the block naming has been evaluated in our previous work [13]. We have compared our block names with names that are manually given by three domain experts with more than ten years of experience working on Eclipse development (see [14]). The result of the comparison shows that more than 70% of names are the same.

Figure 12 presents the feature dependencies in the Eclipse input variants. Each node represents an identified feature, the size of a node is related to the number of elements in that feature. Edges correspond to features dependencies.

-Precision, Recall, and F1-Score Results:. We depicted in Figure 13 the obtained values of `precision` and `recall` for the first dataset (official Eclipse 2020-03 R variants).

In this Figure, we have classified the scores on two categories: i) scores for features that contain more than three plugins (40 features). ii) scores for features that contain less than three plugins (36 features). As we can see, for 78% (31/40) of features in the first category, we obtained `precision` scores greater than 75%. And, 88% (35/40) of these features have `recall` scores greater than 75%.

Furthermore, the obtained `precision` and `recall` scores for all the features that contain more than 60 plugins are greater than 85%. These features represent in general the "core components" of the Eclipse projects. Examples of these features include `Eclipse core equinox` ("`Feature 0`"), `Eclipse core tracecompass`, `Eclipse EMF sirius`, `Eclipse ptp core`, `Eclipse passage lic`, and `Eclipse Core jpt`. In particular, for the `Feature 0`, the `precision` = 0.91, `recall` = 0.93, and `F1-Score`= 0.92. This means that, our tool can identify the core features with a very low error rate.

As for the features of the second category, most of them have very bad precision and recall scores. These scores are explained by the fact that these features are not present in the corresponding predefined Eclipse features files (ground-truth). Actually, some plugins in the Eclipse projects are not categorized in any feature and their feature files are not created by the owners. Therefore, by using our approach these plugins are identified and grouped into features but their scores are very low. To confirm our intuition, we have performed a manual check for each of these features. We found that most of their plugins are present in `plugins folders` of the variants that are containing these features.

[8] Downloaded from: `https://github.com/csytang/LoongFMR`

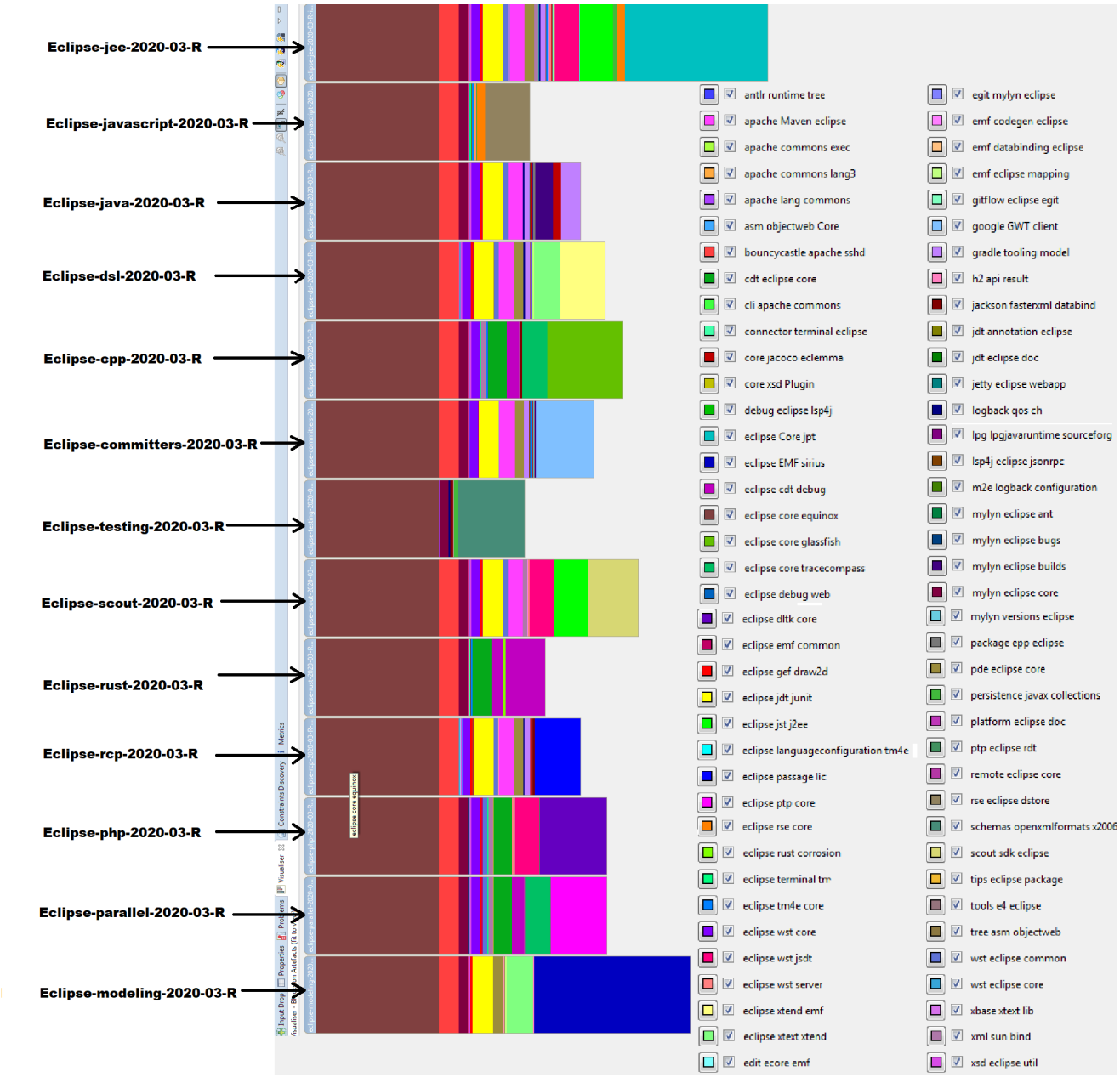

Figure 10: Features per Variant in the Eclipse 2020-03 R Variants

Table 1: Excerpt of distribution of identified features over the Eclipse IDE variants

	Eclipse modeling Variant	Eclipse parallel Variant	eclipse php Variant	Eclipse rcp Variant	eclipse rust Variant	Eclipse scout Variant	Eclipse testing Variant	Eclipse committers Variant	Eclipse cpp Variant	Eclipse dsl Variant	Eclipse java Variant	Eclipse javascript Variant	Eclipse jee Variant
eclipse core equinox	X	X	X	X	X	X	X	X	X	X	X	X	X
bouncycastle apache sshd	X	X	X	X	X	X		X	X	X	X	X	X
apache lang commons	X	X	X		X	X	X	X	X	X	X	X	X
mylyn eclipse core	X	X	X		X	X	X	X	X		X	X	X
mylyn eclipse bugs	X	X	X		X	X	X		X		X	X	X
wst eclipse core		X	X	X		X		X	X	X	X	X	X
mylyn eclipse ant	X					X					X		X
eclipse xtext xtend	X									X			
eclipse core tracecompass		X							X				
eclipse EMF sirius	X												
eclipse ptp core		X											
eclipse passage lic				X									
eclipse xtend emf										X			
eclipse Core jpt													X

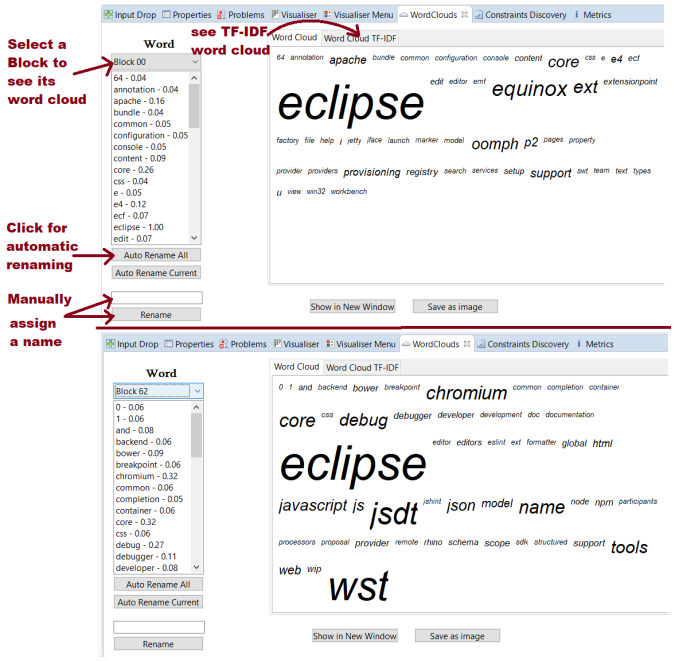

Figure 11: Word clouds show relevant names to two identified Blocks during feature identification

In order to have a global result, we measured the `mean` (average value) and the `median` (obtained by arranging all scores from smallest to largest and locating the central number) scores of all the features as follows:

- `Precision` scores: `mean` = 0.55 and `median` = 0.66
- `Recall` scores: `mean` = 0.54 and `median` = 0.78
- `F1-Score` scores: `mean` = 0.54 and `median` = 0.71

We can observe that the `mean` scores are relatively low. For the `recall` and `F1-Score`, the obtained `median` score is relatively good (50% of features have a score greater than 0.78) compared to the `precision` score.

In addition, we have also estimated the developer's effort when she/he want to correct the identified features. We have analyzed for all of them the False Positives (**FP**) and False Negatives(**FN**). We have observed that 30% of features have "**FP = 0**", 50% of features have "**FP ≤ 3**", and 10% of features "**FP ≤ 11**". These scores mean that for most of the features, the effort that the developer should make to remove manually the unnecessary plugins is not important.

We have also observed that, 68% of features have "**FN = 0**" and 21% of features have "**FN ≤ 12**". These scores show that using our approach, the developer is not requested to make a great effort for adding the missing plugins.

Besides, when we calculated the `mean` and `median` scores for only the features that contain more than three plugins we have obtained the following scores:

- `Precision` scores: `mean` = 0.81 and the `median` = 0.97
- `Recall` scores: `mean` = 0.84 and the `median` = 0.95
- `F1-Score` scores: `mean` = 0.82 and `median` = 0.96

These scores are relatively good compared to the global results.

6.3.2. Results of using the second Dataset

We have used two of the EFLBench strategies for automatic generation of Eclipse IDE variants : i) Random Generation Strategy, ii) Percentage-based Random Generation Strategy. The second strategy allows the user to specify a percentage defining the chances of the features of being selected. In this experimentation, we have used different percentages namely, 20%, 40%, 60%, 80%, and 90%. We have run our experimentation with different set of Eclipse variants: 13 variants (to compare with the scores of the first dateset), 50 variants, and 100 variants (to analyze whether the number of input variants has an impact on feature identification) which are generated automatically.

Table 2 shows the `precision` and `recall` scores that are obtained for the identified features when considering sets of randomly generated Eclipse variants using the *random strategy*. We considered also three configurations 13 variants, 50 variants, and 100 variants.

The table presents for each configuration the `mean` and the `median` scores as well as the number of identified features. As we can see, the obtained `median` values for all the features and for all the configurations is equal to 100%. These mean that at least 50% of these features are correctly identified. As for the `mean` scores for 13 variants, we have obtained scores that are relatively good (78% for precision and 73% for recall) compared to the `mean` scores that are obtained from the 13 official variants (using the first dataset). We can observe here an improvement of around 25%. Thus, using the second dataset, we have obtained in general good scores. This is explained by the fact that the second dataset is generated automatically using a benchmark based on Eclipse. The latter generates the variants starting from one input Eclipse IDE. This means that the number of features with a few number of plugins is reduced (compared to dataset 1) because they are limited only to the features of input Eclipse variant and not to the thirteen Eclipse IDE variants as in the first dataset.

Besides, we can observe how the `mean` precision and recall scores decrease to around 1% to 13% with the number of variants (from 13 to 50 and to 100 variants) which is not very significant. Since the `median` scores have not changed at all for the sets. This means that increasing the number of input variants has almost no impact on the obtained scores. All these values show how is consistent the recovered SAPL using our approach even for a large number of input variants generated randomly.

Furthermore, the sets of randomly generated Eclipse variants using different settings of *Percentage-based Random Strategy* allow to evaluate our approach with variants which are similar, or dissimilar among them. The obtained scores are depicted in Table 3. We have made these observations:

- for the `median` scores, we can see in the table that they are almost 100% in all configurations except the case

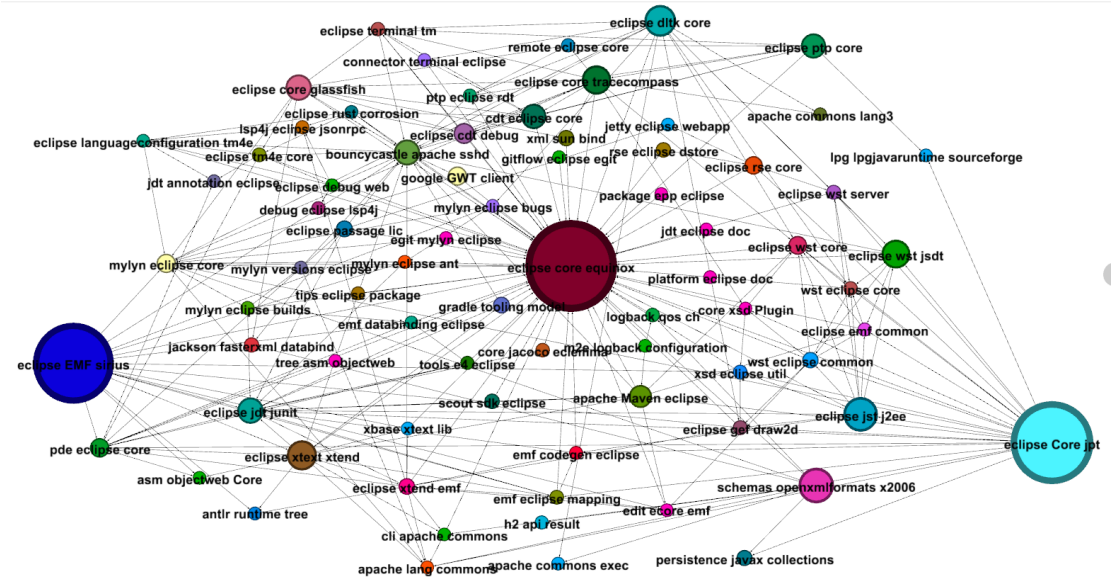

Figure 12: Features dependencies in the Eclipse 2020-03 R Variants.

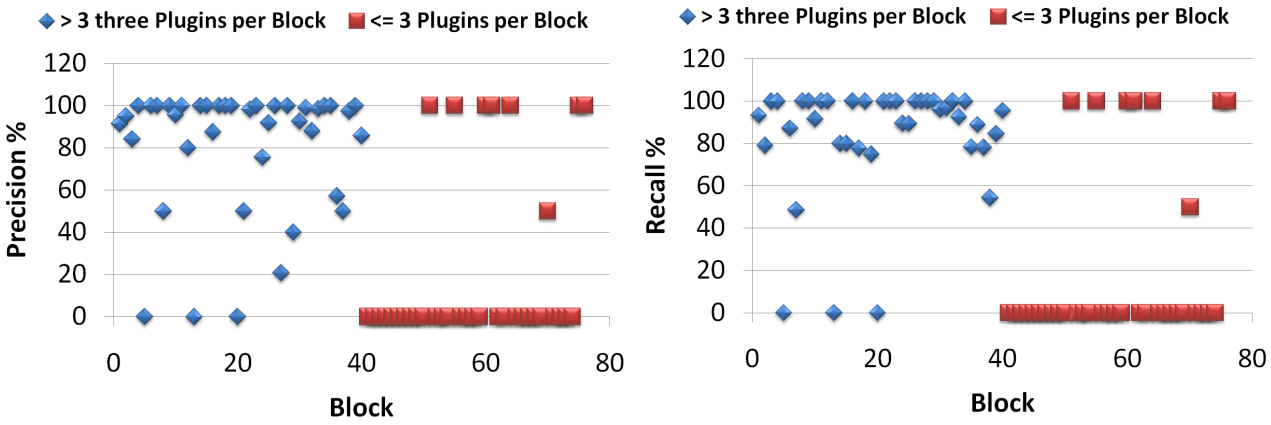

Figure 13: Precision and Recall scores for the Eclipse 2020-03 R variants

Table 2: Precision and Recall Results in sets of randomly generated Eclipse variants using the Random Strategy

	# Variants	Precision	Recall	F1-Score
Mean	13 variants	0.78	0.73	0.75
	50 variants	0.66	0.60	0.62
	100 variants	0.64	0.59	0.61
Median	13 variants	1	1	1
	50 variants	1	1	1
	100 variants	1	1	1
# Features	13 variants	148		
	50 variants	200		
	100 variants	206		

when we set the generation percentage to 20% and the number of input variants is equal to 100.

- for the `mean` scores, as expected the highest scores are obtained when we use a percentage of 90% where almost all the features have been selected from the input Eclipse variant and the generated variants are almost similar. In this case, we obtain a few number of features (see the last three rows in Table 3). In order to explain the scores obtained for the most important features, we take for example the scores of "Feature 0" which contains more than 650 plugins. Its `precision` and `recall` scores are respectively greater than 95% and 90%, which are considered as good values.

- When we set the percentage to 20% and the number of input variants is equal to 50 or 100, the obtained `mean` scores for `recall` and `precision` are relatively low. The lowest values are 60% for `recall` and 66% for `precision`. This comes from the fact that the number of identified features is very high (> 196 features) which leads to low average. But, we have obtained very good *median* scores (greater than 97%) which mean that at least 50% of features have been correctly identified.

Table 3: Precision (Prec) and Recall scores for sets of randomly generated Eclipse variants using different settings of the Percentage(Perc)-based Random Strategy

	# of variants	Perc= 90%		Perc= 80%		Perc= 60%		Perc= 40%		Perc= 20%	
		Prec	Recall	Prec	Recall	Prec	Recall	Prec	Recall	Prec	Recall
Mean	13 var.	0.92	0.97	0.90	0.91	0.96	0.92	0.95	0.91	0.85	0.79
	50 var.	0.97	0.92	0.97	0.95	0.92	0.86	0.84	0.78	0.69	0.63
	100 var.	0.98	0.93	0.95	0.90	0.87	0.81	0.82	0.76	0.66	0.60
Median	13 var.	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	50 var.	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	100 var.	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	**0.97**

# of Features	13 var.	15	18	44	83	119
	50 var.	32	48	89	140	196
	100 var.	38	64	105	154	203

At the end, we can conclude that our approach gives in general good results when we use variants that are generated using both random strategy or percentage-based strategy, with a user-specified percentage, except the case where the percentage equal to 20% and number of generated variants greater than 50.

At the end, based on all the obtained results (using the two datasets), we can conclude that the recovered SAPL is consistent, which shows the effectiveness of our bottom-up process (answer to **RQ1**).

6.4. Answering Research Question 2

In this part of experiment, we used as input only the first dataset (the official Eclipse release that contains 13 variants). In order to evaluate the effectiveness of our derivation process, we have used the SAPL that is recovered from these variants, and our `software architecture composer` to derive "*9 customized variants*". After that, they are compared with the variants that are created manually as explained above. Each "*customized-variant*", is composed of features that are identified from a given input variant (Eclipse IDE variant) and single additional feature. This latter should be identified from other input variants. We describe in Table 4 these "*9 customized variants*". The features that are identified from the candidate variant in the second column are composed with the feature in the third column.

- The `Eclipse EMF Sirius` feature that is identified using our adapter belongs **only** to the `Eclipse-Modeling Tools` variant. It is an Eclipse project which allows to create graphical modeling workbenchs by leveraging the Eclipse Modeling technologies, including EMF and GMF.
- The `Eclipse Core Tracecompass` feature is identified from two variants: `Eclipse-Parallel` and `Eclipse-CPP`. The `Tracecompass` is an open source project to solve performance and reliability issues by reading and analyzing the traces and logs of a system [35].
- The `Eclipse PTP Core` feature is identified from the `Eclipse-Parallel` variant. The PTP project provides an integrated development environment to support the development of parallel applications written in C, C++, and Fortran [36].

In order to select the features that are identified from the candidate variant, we have used the table that was generated in during our bottom-up process (such as Table 1) and show the distribution of identified features over the input Eclipse IDE variants. We have estimated time spent to perform this a "mechanical" task that does not imply thinking. The estimated value is less than 5 minutes. We estimated the time spent for selecting the candidate feature and checking automatically the satisfaction of all the constraints, and then derive the corresponding variant. All these tasks take around 10 minutes which is not significant.

Table 4: Description of the Customized Variants

Custom Variants	Candidate Input Variants	Added Feature	Identified from Variants
CV1	Eclipse-Java	**Eclipse EMF Sirius**	- Eclipse Modeling Tools
CV2	Eclipse-JEE		
CV3	Eclipse-DSL		
CV4	Eclipse-Java	**Eclipse Core Tracecompass**	- Eclipse-Parallel for Scientific Computing - Eclipse-CPP for C/C++ Developers
CV5	Eclipse-JEE		
CV6	Eclipse-Testing		
CV7	Eclipse CPP	**Eclipse PTP Core**	- Eclipse-Parallel for Scientific Computing
CV8	Eclipse-Java		
CV9	Eclipse RCP		

From the other hand, we installed manually (without our approach) each of the three features in the candidate variants by using Eclipse's graphical user interface: clicking on Help->Install New Software...). We add the following URLs to reach the features update sites:

- `http://download.eclipse.org/sirius/updates/releases/6.3.1/2019-06`
- `http://download.eclipse.org/tracecompass/stable/repository/`
- `https://download.eclipse.org/tools/ptp/updates/mars`

At the end, we compared each derived Software Architecture (SA) with the architecture of the corresponding manually configured variant. The obtained `MoJoFM` and `a2a` scores for each derived variant are presented in Table 5. As we can observe, all the values can be considered as good results. They

Table 5: MoJoFM and a2a Results

Custom Variants	MoJoFM	a2a	Number of Plugins	
			Derived SA.	Manually created SA.
CV1	98.41	89.03	1039	722
CV2	97.44	91.26	1479	1196
CV3	99.27	90.80	1021	751
CV4	99.42	96.41	621	566
CV5	97.43	98.45	1099	1071
CV6	98.71	93.58	568	487
CV7	99.81	94.43	736	706
CV8	99.42	96.57	720	722
CV9	99.38	95.83	798	668

indicate that all derived architectures are almost the same with their corresponding variants that are constructed manually. We explain the difference between them as follows. When the size (number of plugins) of the derived variant is greater than the size of the input variant (e.g. CPP, JavaScript, Modeling, etc.), this is due to the fact that in the derived variant there are some plugins (which belong to other features) that have been added for meeting the constraints that are discovered using BUT4Reuse. For example, the high number of plugins in the first customized variant (CV1: Sirius + Eclipse Java features) is due to the set of constraints applied when we select the `Eclipse EMF Sirius` feature. Example of these constraints include:

- "`Eclipse EMF sirius` *implies* `Eclipse xtext xtend`"
- "`Eclipse EMF sirius` *implies* `mylyn eclipse ant`"
- "`Eclipse xtext xtend` *implies* `Eclipse xtend emf`"

where the features "`Eclipse xtext xtend`" and "`Eclipse xtend emf`" are not features of the official `Eclipse-Java` variant and they are not installed during the manual installation of the `Eclipse EMF sirius` feature. But, based on the results of constraints identification (see previous constraints), we observed that they are required for creating a correct variant, while the manual installation of the feature did not do it. In practice, developers should deal with this issue by manually finding and adding the required plugins, which is a complex, time-consuming and error-prone task. In the second customized variant (CV2: Sirius + Eclipse JEE features), the scores are slightly improved. This is due to the fact that the required features already exist in the candidate official variant (Eclipse JEE). For instance, the "`mylyn eclipse ant`" feature already belongs to features of Eclipse JEE variant.

At the end, we can conclude that the obtained results show the effectiveness of our approach. Indeed, taking into consideration the discovered constraints can make the derived variants more consistent and more pertinent than some variants that are created manually. These are different from those used in the evaluation of *RQ1*, which are ground truth variants that are correct because they have been created by experts and exist since a long time. But for new variants created manually, we can obtain incorrect products regarding the SAPL(answer to **RQ2**).

6.5. Genericity Guidelines

For achieving a more generalizable use of the proposed approach, we provide here guidelines for users on how to apply our approach for other families of products than Eclipse-based variants. Let us take the example of Java's module system [37] as a candidate Component Model. In a Java module-based application, the primitive components are called modules. A module is a collection of classes and packages that make up a complete whole. It can be in the form of a directory or a JAR file. A system typically uses multiple modules. Modules were introduced to allow better modularity of the Java platform. To reuse our approach, the developers can follow these guidelines:

1. Mapping our SAPL Metamodel to the candidate Component Model (for instance, Java Modules):
   - First, identify the Elements that compose an artifact in the input variant (e.g., JavaModuleElement and JavaConnectorElement). This will define the granularity of the elements in a given artifact type.
   - Second, identify the points of view that are related to abstraction aspects. For the Java module system, the views could be defined as: ServiceElement, InterfaceElement, PackageElement and ModuleElement.
2. Modify the implementation of our software architecture adapter as follows:
   - Create a Java class for each identified element.
   - Create a Java class that performs the adaptation task (for instance, `JavaModuleSoftArchiAdpater`). It should implements the operations `isAdaptable`, `adapt`, and `construct` of the Java Interface `IAdapter`. The developers can use the implementation of our `OSGiSoftArchiAdapter` class. The result of adaptation is an `AdaptedModel`.
   - Create a Java class that performs the SAPL construction by using as input the "`AdaptedModel`". This class should implement the operation: `createSAPLModel` that is provided by our Java interface: `ISoftwareArchitectureProductLineSynthesis`.
3. Use GMF Dashboard Framework to customize our graphical tool in order to visualize Java Module SAPL.

6.6. Threats to Validity

This experiment may suffer from some threats to the validity of its results:

On what concerns the internal validity, we can say that, as authors have been involved in the manual installation of new customized variants, the results would be biased. In fact, the way of installing new features in a given Eclipse variant was performed by following the official online documentation of the candidate projects (such as `EMF Sirius` , or `CoreTracecompass`). Knowing the way of how installing manually these features does not impact the automatic derivation of such customized variants using our composer, because the developer has only to

select the candidate features and the required features are automatically selected and added by meeting the constraints that are identified at the SAPL recovering step.

Besides, we have used in our evaluation the predefined Eclipse *features* as `ground truth` to compare them with the features that are identified using our approach. The fact that the predefined *features* are realized manually by the Eclipse Projects owners can be seen as a threat to validity, because some plugins in the Eclipse projects are not categorized in any feature and their feature files are not created by the owners. Therefore, by using our approach these plugins are identified and grouped into features but their scores are very low. But, the most important features in the Eclipse projects which are well described by the owners are identified using our approach with high `precision` and `recall` scores.

The fact that we have instantiated and experimented our approach in the context of Eclipse-based SPL can be seen as a limit for our study's generalizability to other kind of component/ service-based SPL. To mitigate this, we provided in our approach a generic meta-model for component-based software variants which can be easily instantiated for other kinds of component/service based software variants. At the implementation level, our plugin provides extension-points for other developers to contribute by developing extensions for generating SAPL for other kinds of component-based software product line, such as applications built with Java 9+ module system.

The fact that Eclipse IDE variants are very well modularized can be seen as a threat for the obtained results in our experimentation. In fact, using software variants created with opportunistic reuse with poor modularization has no effect on the efficiency of our architecture extraction approach. But, it may extract complex and highly coupled architecture. However, the aim of our approah is also to enable the developer to construct a multi-point of view SAPL. These points of view can help and assist the developer to understand progressively the SAPL and reduce the complexity of the created architecture. Indeed, it is not easy to understand the whole system by analyzing all the points of view mixed-up together.

To increase construct validity, we did not limit ourselves to a single evaluation measure. Indeed, for measuring the performance of the SAPL reconstruction process, our evaluation was conducted using three measures (`recall`, `precision`, and `F1-Score`), which are widely accepted in software engineering research community [38]. In addition, for measuring the accuracy of the derived software architectures using our approach, we used two well known architectural change metrics: *Architecture2Architecture* (`a2a`) [6] and `MoJoFM` [34].

7. Related Work

Assunção et al. [22] presented a complete survey on the existing SPL adoption approaches. They exposed three ways for adopting SPLE: i) from scratch, by applying a complete domain analysis and variability management before application engineering ii) by creating and updating the SPL when every new product appears; and iii) by using an extractive approach, which takes existing products as the basis for the core assets. The extractive approach is also called SPL reengineering [11].

Our approach is built on BUT4Reuse where a new adapter is proposed to consider software architectures. Independently from SA, several extensions of BUT4Reuse have already been developed and published in [21, 24, 39]. Martinez et al. [21] proposed an approach for automating the extraction of model-based SPL from model variants as follows. First, they identify features and detect constraints among them. After that, the model variants are refactored to conform to an SPL approach. Ziadi and Hillah [39] proposed an adapter for BUT4Reuse to extract variability from Bytecode based applications. The work in [40] is designed to identify variability in a set of statechart variants. They have used variability mining algorithms to identify the relations between the variants. In our approach, we first reverse-engineer the software architecture of each software variant, after that, we reconstruct a SAPL model starting from which the software architecture of each new software variant can be effectively derived.

Besides, software architecture recovery (SAR) is a challenging problem, and several works in the literature have already proposed contributions to solve it (e.g., works cited in [41, 6, 42]). Most of these approaches are proposed for a single software architecture recovery. Lutellier et al. [6] present a comparative analysis of six SAR techniques. Maqbool et al. [42] presented a review of the hierarchical clustering techniques. In the last decade several works had proposed approaches that aim to recover component-/service-oriented architectures from existing systems. For example, the works in [43] and [44] are based on the definition of a correspondence model between the code elements and the architectural concepts. In [45, 46] a component is considered as a group of classes collaborating to provide a system function. Seriai et al. in [47] used FCA to perform the component interface identification. The authors in [48] recover BPMN models starting from service oriented systems that have been generated from web applications. More recently, Shatnawi et al. [49] proposed a new approach to extract reusable services from the source code of a collection of software variants. Similar to BUT4Reuse principals, this approach is also based on the comparison of input applications to identify and cluster similar services. Some works have been proposed to recover software architectures at run-time. For instance, [50] presented an approach for recovering at run-time software architectures from component based systems and changing the system via manipulating the recovered SA. The authors in [51] have proposed an approach to recover at run-time architectures of a large-sized component/service oriented systems by considering some specific use cases in order to reduce the complexity of the recovered architectures. Compared to our work, we focused on the software architecture recovery for a family of products not only a single software architecture recovery.

Our approach is extensible by allowing to use one of the existing approaches for recovering software architecture of each software variant. Regarding SAPL recovery, it is also extensible. However, the organization of features in the recovered SAPL model is based on the result of the feature identification and constraint discovering. The BUT4Reue framework allows

to extend easily this activity by implementing one of the existing approaches such as FCA.

Besides, few works were proposed in the literature that aim to recover SAPL models. The authors in [12] have presented a mapping study of the existing approaches of software architecture recovery for software product line. Shatnawi et al. [5] have proposed a process for recovering software product line architectures of a family of object-oriented product variants. First, they used FCA to migrate the object-oriented systems to a set of component variants. Each variant is a set of similar components that share the majority of their classes and dependencies. Second, they used FCA to identify mandatory and optional components. At the end, they build the SPLA as a feature model where the dependencies between component variants are based on relations of type *alternative*, OR, AND, *require* and *exclude*. The authors in [4] have proposed an approach for recovering software product line architecture from object-oriented product variants. They identify mandatory components and variation points of components as a main step. They analyze commonality and variability across product variants in terms of features.

Compared to our work, the recovered SAPL using our approach is both a feature model and a complete architecture that shows all the architectural connections between components. The particularity for us is that the software architectures are considered as main artifacts. We believe that recovering SAPL for such software architecture variants helps the developers not only the derivation of new variants but also in the maintenance and evolution of the family of products. Indeed, this SAPL allows to have a specific documentation for each of the variants and therefore to be able to maintain and evolve independently.

Wille et al. [52] have proposed a variability mining approach for Technical Architecture (TA) variants. They eliminate the unnecessary information from the input TAs. The components from the TAs are clustered by filtering them based on their structural relations to eliminate unrealistic variability. Unfortunately, their approach can not recover an architecture describing all the variants. On the other hand, our solution can derive new SAs and product variants starting from the reconstructed SAPL. The proposed process is generic and can be applied for many component based-systems (or -software architectures).

Assunção et al. [53] proposed an approach called ModelVars2SPL. It allows to extract starting from UML class diagram variants a feature model and a product line architecture (PLA) which represents a global structure of the input variants. The input of ModelVars2SPL consists of two parts: i) a set of model variants, and ii) for each variant a feature set that denotes the configuration of the features provided by the variant. Compared to our work, we identify automatically the features from input variants. Moreover, the generated SAPL using our approach is both a feature model and an architecture where each feature has it owns fragment of architecture. For large and complex input models variants, our SAPL enables to make the understanding progressive (per feature) which is more easy than analyzing the complete architecture. In addition, our approach is generic and can be applied on a variety of models (not only UML class ones).

The authors in [54, 55] have proposed an approach for recovering a product line architecture (PLA) starting from a set of source code variants. Their approach supports the identification of a minimum subset of cross-product architectural information through the identification and removal of outliers. The recovered PLA using their approach is presented using UML package and class diagrams, module dependency graphs and design structure matrices. The UML diagrams are annotated to highlight the commonality and variability related to assets. Another approach has been proposed recently by Lee et al. in [56] which aims to recover a PLA starting from a family of products developed with the clone-and-own approach. The PLA is an abstraction of all possible product variants of an SPL. For determining common and variable classes, they use the Harmonized Total Constant Commonality Indices (HTCCIPL) of packages or classes of a product line. They applied the approach on set of Apo-Games variants. The PLA in their work is presented as a package diagram or class diagram annotated with HTCCIPL values.

Compared to our approach, their approach applies on small- to medium-sized software variants where the variability is identified at low level artifacts. In addition, the derivation of new software architecture variants starting from their recovered PLA has not been addressed.

The authors in [57] have compared in term of precision and recall five search algorithms to locate features over families of product models guided by latent semantic analysis (LSA), a technique that measures similarities between textual queries. Their results show that search-based software engineering (SBSE) techniques can be applied to locate features in product models. Recently machine learning techniques are widely used for feature location process [58, 59, 60]. The authors in [58] proposed a machine learning-based approach for feature location on models. The goal is to identify the model fragments that best realizes specific features. They use different subsets of a knowledge base to learn how to locate unknown features. Their approach analyze the influence of three model fragment properties: density, multiplicity, and dispersion. Density measures the percentage of model elements that are present in a model fragment. Multiplicity measures the number of times that the model fragment appears in the model. Dispersion measures the ratio of connected elements in the model fragment (Model elements may or may not be connected in the model). Their results show that density and dispersion properties significantly influence the feature location results. In our approach, thanks to the extensibily mechanism of BUT4Reuse Framework, we argue that it is possible implement and improve the feature identification process by using one of existing machine learning approach.

8. Conclusion

Recovering architecture models of large-sized software products is an important activity in software maintenance and evolution. These architecture models offer a good documentation to understand the software product before changing it. For large software products with several software variants, these models are of great interest since they enable also to see the common

and variable features between software variants. SPL Reverse Engineering (SPL-RE) processes enable to recover models with such a rich structure, including the variable part in the product variants and enable to see the variability points.

In our work, we focused on component- and service-based software variants and proposed in this paper: i) a (meta-)model for architectures of component/service-based software product line which describes the rules for defining an SAPL, ii) the design of a generic SPL-RE process for building architecture models (SAPL models) by analyzing software variants, iii) a forward engineering process that uses the recovered SAPL to derive new consistent and useful software architecture variants, iv) an implementation of the approach in the context of OSGi-based systems, and v) an experimentation of this approach on a set of Eclipse releases. The experimentation that we conducted enabled us to evaluate the efficiency of the process in identifying correct features, compared to those identified/built by experts. In addition, it enabled us to measure the accuracy of architectures of products derived from the recovered SAPL.

At the implementation level, we have proposed in this work a tool chain completely based on Eclipse Modeling Framework, Graphical Modeling Framework and FeatureIDE Framework [17]. It consists of: i) a graphical tool for creating SAPL and defining constraints between features, ii) a graphical tool for creating and updating each feature's software architecture, iii) a new BUT4Reuse adapter related to software architecture variants. The adapter is designed with a set of parameters to consider different architectural points of view (services, interfaces, packages and extensions), iv) a FeatureIDE Software Architecture Composer for selecting one possible configuration and checking if it satisfies all the constraints and than deriving the corresponding software architecture.

As perspectives to this work, we plan to study the enrichment of SPL reverse engineering of large component/service-based variants by including a learning module which exploits existing SPLs and their variants/features to identify features in a smarter way (by learning from existing experiences). In addition, we envisage the instantiation of the process for other component/service frameworks, or just investigate its use with Java modules for exploring variability in standard Java applications. Our approach is applicable on any set of large software product variants: i) other IDEs (from Jetbrains or Microsoft for instance), ii) Linux distributions, iii) existing code bases of companies from which they used to derive new products for their new customers, and many other software development settings. This requests of course a set of adapters to be developed for each case, but the general philosophy of our approach remains the same.

References

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

[2] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, T. Leich, FeatureIDE: An extensible framework for feature-oriented software development, Science of Computer Programming 79 (0) (2014). doi:http://dx.doi.org/10.1016/j.scico.2012.06.002.

[3] J. Martinez, T. Ziadi, M. Papadakis, T. F. Bissyandé, J. Klein, Y. Le Traon, Feature location benchmark for extractive software product line adoption research using realistic and synthetic eclipse variants, Information and Software Technology 104 (2018) 46–59.

[4] H. Eyal-Salman, A.-D. Seriai, Toward recovering component-based software product line architecture from object-oriented product variants, in: Proc. of SEKE, 2016, pp. 1–7.

[5] A. Shatnawi, A.-D. Seriai, H. Sahraoui, Recovering software product line architecture of a family of object-oriented product variants, J. Syst. Softw. (JSS) 131 (C) (2017) 325–346.

[6] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidović, R. Kroeger, Measuring the impact of code dependencies on software architecture recovery techniques, IEEE TSE 44 (2) (2018) 159–181.

[7] L. Bass, P. Clements, R. Kazman, Software architecture in practice, 3rd edition, Addison-Wesley Professional, 2012.

[8] B. P. Lientz, E. B. Swanson, Software Maintenance Management, Addison Wesley, Reading, MA, 1980.

[9] J. Garcia, I. Ivkovic, N. Medvidovic, A comparative analysis of software architecture recovery techniques, in: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2013, pp. 486–496.

[10] J. Martinez, W. K. Assunção, T. Ziadi, Espla: A catalog of extractive spl adoption case studies, in: Proceedings of the 21st International Systems and Software Product Line Conference-Volume B, 2017, pp. 38–41.

[11] W. K. Assunção, R. E. Lopez-Herrejon, L. Linsbauer, S. R. Vergilio, A. Egyed, Reengineering legacy applications into software product lines: a systematic mapping, Empirical Software Engineering 22 (6) (2017) 2972–3016.

[12] Z. T. Sinkala, M. Blom, S. Herold, A mapping study of software architecture recovery for software product lines, in: Companion Proceedings of ECSA, 2018, pp. 1–7.

[13] M. L. Kerdoudi, T. Ziadi, C. Tibermacine, S. Sadou, Recovering software architecture product lines, in: 2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS), IEEE, 2019, pp. 226–235.

[14] 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: Proc. of SPLC, Nashville, TN, USA, 2015, pp. 101–110.

[15] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, Y. L. Traon, Bottom-up technologies for reuse: automated extractive adoption of software product lines, in: Proc. of ICSE Companion, IEEE Press, 2017, pp. 67–70.

[16] J. Martinez, Mining software artefact variants for product line migration and analysis, Ph.D. thesis, University Pierre et Marie Curie and University of Luxembourg (2016).

[17] T. Thüm, T. Leich, S. Krieter, Feature modeling and development with featureide, Modellierung 2018 (2018).

[18] L. Northrop, P. Clements, F. Bachmann, J. Bergey, G. Chastek, S. Cohen, P. Donohoe, L. Jones, R. Krut, R. Little, et al., A framework for software product line practice, version 5.0, SEI.–2007–http://www. sei. cmu. edu/productlines/index. html (2007).

[19] K. Pohl, G. Böckle, F. J. van Der Linden, Software product line engineering: foundations, principles and techniques, Springer Science & Business Media, 2005.

[20] D. Benavides, S. Segura, A. Ruiz-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.
URL http://dx.doi.org/10.1016/j.is.2010.01.001

[21] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, Y. L. Traon, Automating the extraction of model-based software product lines from model variants (T), in: 30th IEEE/ACM, ASE , Lincoln, NE, USA,, 2015, pp. 396–406.

[22] W. K. G. Assunção, S. R. Vergilio, Feature location for software product line migration: a mapping study, in: 18th SPLC, Companion Volume, Italy, 2014, pp. 52–59.

[23] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, Y. L. Traon, Bottom-up technologies for reuse: automated extractive adoption of software product lines, in: Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume, 2017, pp. 67–70.

[24] L. Li, J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, Y. L. Traon, Mining families of Android applications for extractive SPL adoption, in: Proceedings of the 20th SPLC 2016, Beijing, China, 2016, pp. 271–275.

[25] M. L. Kerdoudi, T. Ziadi, C. Tibermacine, S. Sadou, A bottom-up approach for reconstructing software architecture product lines, in: Proc. of the 13th ECSA: Companion Proceedings, ACM, 2019, pp. 46–49.
[26] J. McAffer, P. VanderLei, S. Archer, OSGi and Equinox: Creating highly modular Java systems, Addison-Wesley Professional, 2010.
[27] G. Perrouin, J. Klein, N. Guelfi, J.-M. Jézéquel, Reconciling automation and flexibility in product derivation, in: Proc of. the 12th SPLC, IEEE, 2008, pp. 339–348.
[28] T. Ziadi, L. Frias, M. A. A. da Silva, M. Ziane, Feature identification from the source code of product variants, in: 16th CSMR 2012, Hungary, 2012, pp. 417–422. doi:10.1109/CSMR.2012.52.
[29] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, Y. L. Traon, Name suggestions during feature identification: The VariClouds approach, in: Proceedings of the 20th SPLC , Beijing, China, 2016, pp. 119–123.
[30] P. B. Kruchten, The 4+ 1 view model of architecture, IEEE software 12 (6) (1995) 42–50.
[31] A. Bryan, J. Ko, S. Hu, Y. Koren, Co-evolution of product families and assembly systems, CIRP annals 56 (1) (2007) 41–44.
[32] J. C. Kirchhof, M. Nieke, I. Schaefer, D. Schmalzing, M. Schulze, Variant and product line co-evolution, in: Model-Based Engineering of Collaborative Embedded Systems, Springer, 2021, pp. 333–351.
[33] C. Seidl, F. Heidenreich, U. Aßmann, Co-evolution of models and feature mapping in software product lines, in: Proceedings of the 16th International Software Product Line Conference - Volume 1, SPLC '12, Association for Computing Machinery, New York, NY, USA, 2012, p. 76–85. doi:10.1145/2362536.2362550.
URL https://doi.org/10.1145/2362536.2362550
[34] Z. Wen, V. Tzerpos, An effectiveness measure for software clustering algorithms, in: Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004., IEEE, 2004, pp. 194–203.
[35] E. Foundation, Eclipse trace compass project, https://www.eclipse.org/tracecompass/.
[36] E. Foundation, Eclipse parallel tools platform (ptp), https://www.eclipse.org/ptp/.
[37] S. Mak, P. Bakker, Java 9 Modularity: Patterns and Practices for Developing Maintainable Applications, " O'Reilly Media, Inc.", 2017.
[38] G. Salton, M. J. McGill, Introduction to Modern Information Retrieval, McGraw-Hill, Inc., New York, NY, USA, 1986.
[39] T. Ziadi, L. M. Hillah, Software product line extraction from bytecode based applications, in: Proc. of the 23rd (ICECCS), IEEE, 2018, pp. 221–225.
[40] D. Wille, S. Schulze, I. Schaefer, Variability mining of state charts, in: Proceedings of the 7th International Workshop on Feature-Oriented Software Development, 2016, pp. 63–73.
[41] S. Ducasse, D. Pollet, Software architecture reconstruction: A process-oriented taxonomy, IEEE TSE 35 (4) (2009) 573–591.
[42] O. Maqbool, H. Babri, Hierarchical clustering for software architecture recovery, IEEE TSE 33 (11) (2007) 759–780.
[43] S. Chardigny, A. Seriai, M. Oussalah, D. Tamzalit, Extraction of component-based architecture from object-oriented systems, in: Proc. of WICSA, IEEE, 2008, pp. 285–288.
[44] A. Seriai, S. Sadou, H. A. Sahraoui, Enactment of components extracted from an object- oriented application, in: Proc. ECSA, Springer, 2014, pp. 234–249.
[45] S. Allier, H. A. Sahraoui, S. Sadou, S. Vaucher, Restructuring object-oriented applications into component-oriented applications by using consistency with execution traces, in: Proc. of the 13th CBSE'10, Springer, 2010, pp. 216–231.
[46] S. Allier, S. Sadou, H. A. Sahraoui, R. Fleurquin, From object-oriented applications to component-oriented applications via component- oriented architecture, in: Proc. of the 9th WICSA, Colorado, USA, IEEE, 2011, pp. 214–223.
[47] A. Seriai, S. Sadou, H. Sahraoui, S. Hamza, Deriving component interfaces after a restructuring of a legacy system, in: Proc. of WICSA, IEEE, 2014, pp. 31–40.
[48] M. L. Kerdoudi, C. Tibermacine, S. Sadou, Opening web applications for third-party development: a service-oriented solution, Journal of SOCA 10 (4) (2016) 437–463.
[49] A. Shatnawi, A. Seriai, H. A. Sahraoui, T. Ziadi, A. Seriai, Reside: Reusable service identification from software families, J. Syst. Softw. (JSS) 170 (2020) 110748. doi:10.1016/j.jss.2020.110748.
URL https://doi.org/10.1016/j.jss.2020.110748
[50] G. Huang, H. Mei, F.-Q. Yang, Runtime recovery and manipulation of software architecture of component-based systems, Journal of ASE 13 (2) (2006) 257–281.
[51] M. L. Kerdoudi, C. Tibermacine, S. Sadou, Spotlighting use case specific architectures, in: Proc. the 12th ECSA, Springer, 2018, pp. 236–244.
[52] D. Wille, K. Wehling, C. Seidl, M. Pluchator, I. Schaefer, Variability mining of technical architectures, in: Proceedings of the 21st SPLC - Volume A, ACM, 2017, pp. 39–48.
[53] W. K. Assunção, S. R. Vergilio, R. E. Lopez-Herrejon, Automatic extraction of product line architecture and feature models from uml class diagram variants, Information and Software Technology 117 (2020) 106198.
[54] C. Lima, I. Machado, M. Galster, C. von Flach G. Chavez, Recovering architectural variability from source code, in: Proceedings of the 34th Brazilian Symposium on Software Engineering, 2020, pp. 808–817.
[55] C. Lima, W. K. Assunção, J. Martinez, I. do Carmo Machado, C. von Flach G. Chavez, W. D. Mendonça, Towards an automated product line architecture recovery: the apo-games case study, in: Proceedings of the VII Brazilian Symposium on Software Components, Architectures, and Reuse, 2018, pp. 33–42.
[56] J. Lee, T. Kim, S. Kang, Recovering software product line architecture of product variants developed with the clone-and-own approach, in: 2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC), IEEE, 2020, pp. 985–990.
[57] J. Font, L. Arcega, Ø. Haugen, C. Cetina, Achieving feature location in families of models through the use of search-based software engineering, IEEE Transactions on Evolutionary Computation 22 (3) (2017) 363–377.
[58] M. Ballarín, A. C. Marcén, V. Pelechano, C. Cetina, On the influence of model fragment properties on a machine learning-based approach for feature location, Information and Software Technology 129 (2021) 106430.
[59] C. S. Corley, K. Damevski, N. A. Kraft, Exploring the use of deep learning for feature location, in: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2015, pp. 556–560.
[60] A. C. Marcén, J. Font, Ó. Pastor, C. Cetina, Towards feature location in models through a learning to rank approach, in: Proceedings of the 21st International Systems and Software Product Line Conference-Volume B, 2017, pp. 57–64.