Recovering Software Architecture Product Lines

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

Recovering Software Architecture Product Lines

Mohamed Lamine Kerdoudi
Computer Science Department
Biskra University, Algeria
Email: L.Kerdoudi@univ-biskra.dz

Tewfik Ziadi
Sorbonne Université
CNRS, LIP6, F-75005 Paris, France
Email: Tewfik.Ziadi@lip6.fr

Chouki Tibermacine
LIRMM, CNRS
and Montpellier University, France
Email: Chouki.Tibermacine@lirmm.fr

Salah Sadou
IRISA- University of South Brittany, France
Email: Salah.Sadou@irisa.fr

***Abstract*—A large component and service-based 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. However, for deriving new product 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. Indeed, this enables them to see the invariant part of the whole, on top of which new functionality can be built, in addition to the different options they can use. We investigate in this work the use of software product line reverse engineering approaches, and in particular the framework named But4Reuse, for recovering an architecture model that enables us to build a Software Architecture Product Line (SAPL), from a set of software variants. We propose a generic process for recovering an architecture model of such a product line. We have instantiated this process for the OSGi Java framework and experimented it for building the architecture model of Eclipse IDE SPL. The results of this experimentation showed that this process can effectively reconstruct such an architecture model.**

I. Introduction

Software Product Line (SPL) Engineering (SPLE) considers the existence of a single model describing all the variants that implement each architecture element.

The particularity of this "single" architecture 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. Product variants can be *derived* (generated) by choosing the 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 product derivation [1], [2]. However, there are many systems that exist as several "independent" variants and not as a "single whole". Indeed, large component and service-based software systems exist in different forms, as different variants targeting different business needs and users. For example, IDEs like Eclipse, exist as several variants targeting different kinds of software engineers. These systems often use ad hoc mechanisms to manage variability and they do not take complete benefits from the SPLE framework.

For developers of new product 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. 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 options they can use.

This paper considers the challenge of analysing the source code of existing variants of component and service-based software systems to reverse-engineer a software architecture following that is common to all the existing variants. We call this constructed architecture *Software Architecture Product Line* (SAPL) that represents the unique software architecture that supports the software product line and common to all the product variants members of the SPL.

We defend a vision by considering SAPL as a reference architecture starting from which the architecture of each product 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 product. From the point of view of the responsible on the maintenance of the product, the architecture is a crucial artifact for two reasons: i) understand the product before making the changes, and ii) notify changes made on the product to keep its documentation compliant with its implementation. However, the situation where the product variants do not have any specific architecture raises problems during the maintenance stage of a product on the two points mentioned above: i) referring to a generic architecture to understand a given product is a very difficult task. Knowing that understanding is the most costly activity during maintenance, this will generate considerable additional costs. ii) Modifying a generic architecture, to take into account the modifications made on one of its products, is a task that is not only difficult and error prone, but also with unforeseeable consequences on the other products.

Our vision is that the different products (software) can be created from the same SPL, but must be able to evolve independently and without constraints. So, our approach to solving the two problems mentioned above is that the product line must produce the software architecture of a product, but

not directly the corresponding software. This is what we called Software Architecture Product Line (SAPL).

We propose in this paper a process for SAPL-reverse-engineering. This process extends the BUT4Reuse framework, which is considered as one of the most effective methods for SPL-reverse-engineering [3], [4]. This framework was proposed as a generic and extensible framework for SPL reverse-engineering. We extend BUT4Reuse to SAPL reverse-engineer large component and service-based software systems starting from a collection of their existing variants. The remaining of the paper is organized as follows. In Section II, we expose our SAPL-RE process. In Section III, we present an instantiation of the process for the OSGi component model. We show the results of our experiments in Section IV. We finally discuss the related work in Section V, before concluding the paper in Section VI.

II. A Generic Process for SAPL-Reverse Engineering

Before presenting the proposed process for SAPL Reverse Engineering and variant derivation, we first describe the meta-model supported by our approach.

A. SAPL Metamodel for Component-Based Software Variants

Figure 1 depicts the defined SAPL meta-model which is used for creating an architecture for a set of component-based software variants. We have been inspired in the definition of this meta-model by the feature meta-model in [5]. 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 the component-based systems, it has been defined based on an abstract syntax of a software component model. It is used to represent any kind of component-based system such as OSGI, Spring, etc. A generally accepted view of a software 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 be operations performed by the component. The requirements (`RequireElement` in our meta-model) are needed by the component to produce the provided capabilities.

B. SAPL-RE and Component-based Application derivation process

The goal of the process presented in this paper is to analyze existing product variants to extract a SAPL with an explicit specification of commonality and variability. This architecture can be used to generate new variants using the principles of SPLE. This extraction is one of the most challenging research directions identified in the SPL community. Many SPL extraction approaches have been proposed in the last years. Wesley et al. [6] present a complete survey on these existing works. In this paper, we propose to revisit this problem from the software architecture (SA) perspective.

In this context, we identified five main challenges: 1) How to extract a software architecture from the source code; 2) How to compare the architecture variants to identify the common part and find and name different features; 3) How to construct the SAPL with an explicit specification of the variability at an architectural level; 4) How to simplify and reduce the complexity of the recovered architectures. The extraction should be generic and extensible to support all these different views; 5) Once the SAPL is constructed, one remaining challenge is related to the derivation process. How the SAPL can be used to derive new SA variants?

The overall process of our approach to tackle the identified challenges is illustrated in Figure 2. This process is initially introduced in our previous work [7], which is substantially extended in this paper with a detailed specification and a validation. It is composed of three main activities: 1) Reverse-Engineering of SA variants; 2) SAPL Reconstruction; and 3) Variants Derivation. In the following, we describe each activity.

1) Reverse-Engineering of SA Variants: The first activity in our approach is to use reverse-engineering techniques to extract a software architecture variant from the source code of each software variant. As we will see in the next section, the reverse-engineering of SAs from eclipse variants is based on the analysis of the configuration files and the source code of the different components (plugins).

2) SAPL Construction: In this activity, the different SA variants are analyzed and compared to identify the common part and the different features. As illustrated in Figure 2, this activity extends the BUT4Reuse framework to support architectural artefacts. Indeed, BUT4Reuse [3], [4] was proposed as a generic and extensible framework to identify features from a set of similar artifacts. It is extensible by enabling to add different concrete techniques or algorithms for the relevant steps of feature identification, mining feature constraints, extracting reusable assets, synthesizing and visualizing feature models.

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 the 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 from coarse to fine 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

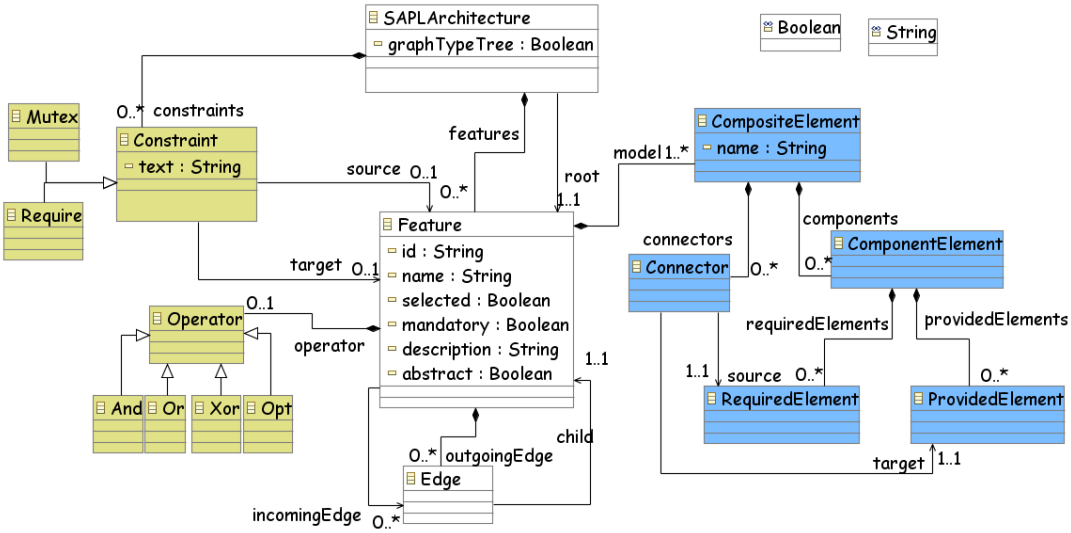

Fig. 1. SAPL Metamodel for Component-Based Software Variants

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 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 software architectures. In addition to allow comparing software architectures, this new adapter is designed with a set of parameters to consider different architectural views (services, interfaces, packages, extensions, etc).

Once the adapter is implemented, SAPL construction follows four sub-activities as illustrated in Figure 2.

a) 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 view.

b) 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*. In this paper, we reused the algorithm called *Interdependent Elements* that formalizes block identification using class equivalences. 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 system. BUT4Reuse integrates what is called *VariCloud* [8], 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 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 we will see in the next section, for our adapter, words correspond to the names of packages, interfaces and plugins.

c) 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.

d) Multi-View SAPL Constrcution: A software architecture of a large system is a complex entity; it cannot be presented in a single view. One of the most important concepts associated with software architectures are views. A view is the result of applying a viewpoint to a particular system of interest (for instance, service-, interface-, and extension-oriented views). In this step of our process, we enable the developer to construct a multi-view SAPL. These views can help and assist the developer to understand progressively the SPL.

3) Variants Derivation: In this step, the developer can select starting from the recovered SAPL a set of features that meet her/his requirements for deriving the architecture of the new variant. We provide a graphical tool to visualize the derived architecture. Once the developer analyzed and understood this architecture, she/he can derive the new product as a new variant.

III. Instantiation of the Process for OSGi Component/Service Model

We have instantiated the previous process for the OSGi Java framework, in order to analyze applications like Eclipse. The OSGi specification defines a component model and a framework for creating highly modular Java systems [9]. Eclipse-based applications run on top of Equinox which is the reference implementation of the OSGi specification. It is a collection of similar software products that share a set

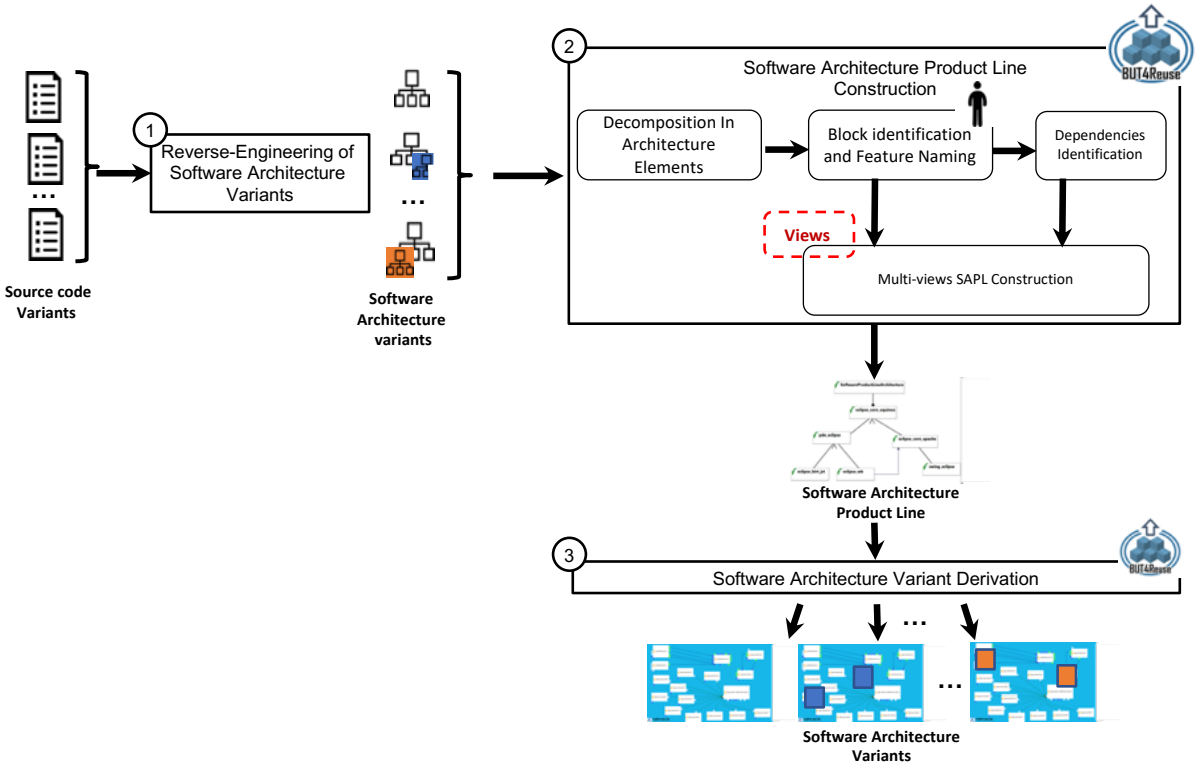

Fig. 2. Proposed SAPL-Reverse Engineering Process

of software assets. It offers a set of "releases" where each one is a large-sized Java application 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 product. The default Eclipse releases are predefined for targeting specific developer needs. Currently, if a developer wants to create a customized release, she/he has to select one of the default releases[1] (for instance, IDE for C/C++ Developers) and then manually install new software which meets her/his requirements. In this paper, we consider Eclipse releases as product variants and we aim to adopt the SAPL approach in order to be able to develop efficiently a personalized Eclipse variant. Before presenting the implementation details, we introduce the OSGi Meta-Model.

A. OSGi Meta-model

Figure 3 presents our metamodel for the OSGi SAPL. It is an adaptation of the meta-model in Figure 1 for the OSGi component model. A component in OSGi is known as a bundle or a plugin (`PluginElement` in this metamodel) 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 metamodel) and defining contracts. Other plugins contribute by developing extensions (`ExtensionElement` in this metamodel) using existing extension points.

Our OSGi meta-model allows to produce several SA views that represent different kinds of plug-in's capabilities and requirements. The supported architecture views in this meta-model are: *interface, service, package, and extension views*. Of course these views are not orthogonal, there are intersections

[1]In https://www.eclipse.org/downloads/packages/release

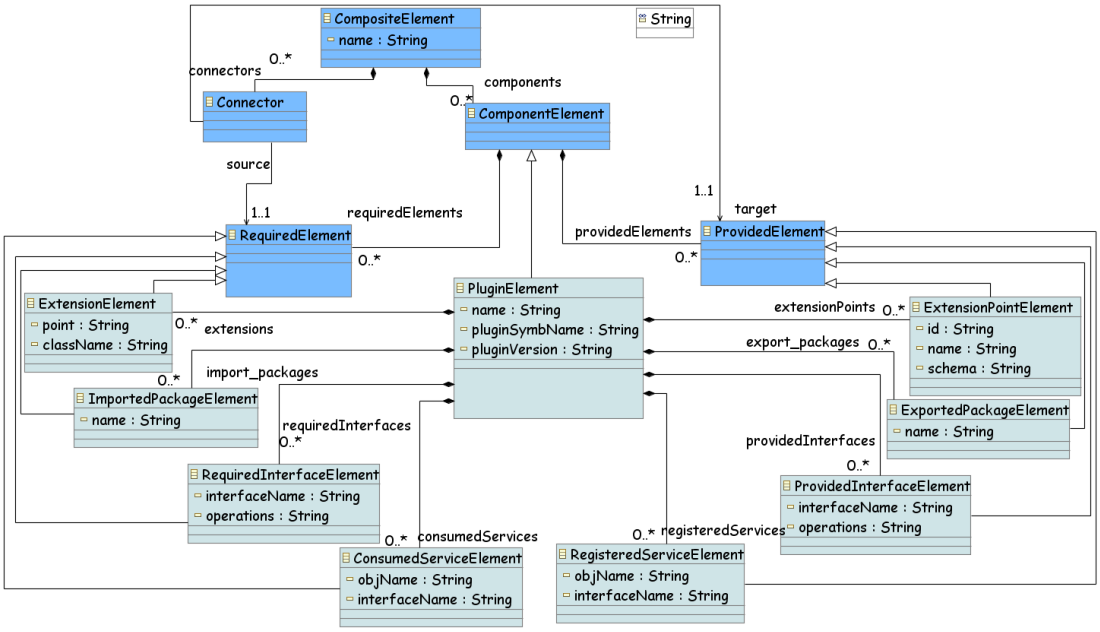

Fig. 3. An OSGi Meta-Model

between each other. But, nobody would be able to understand the whole system by analyzing all the views together. Thanks to this meta-model, developers can progressively understand the system by analyzing each architecture view separately. In addition, our framework can be easily extended to support other views in order to cover all the aspects that the developers need to know when they develop a new variant.

B. Reverse Engineering of the Eclipse Variants

For recovering the SA variants, we analyze the Eclipse artifacts as follows: i) for each variant, we create a `compositeElement` with the name of the variant. ii) for each plug-in, 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 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 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. At the end, we note that the parsing of the source code and bytecode has been implemented by reusing existing Java tools such as AST-Parser and ObjectWeb-ASM.

C. But4Reuse Adapter for Eclipse-Software Architecture Variants

In this section, we present our adapter for Eclipse-SA variants[2]. We have followed the generic activities which are defined in [4] to implement this adapter :

i) **Elements identification**: to compare and analyze several product variants, But4Reuse divides each variant into a set of elements. Our elements are: `PluginElement`, `ServiceElement`, `PackageElement`, `InterfaceElement`, `ExtensionElement`, and `ExtensionPointElement`. 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.

ii) **Similarity strategy**: It involves comparing all pairs of elements of the same kind. Two elements are similar if they have the same name and exactly the same sub-elements. For example, i) two plugin elements are similar, if they have the same symbolic names, and their extension elements and interface elements are also similar. ii) Two interface elements are similar, if they have the same qualified names and contain exactly the same operations (method signatures).

iii) **Structural dependencies identification**: A plugin element structurally depends on its required plugins. These dependencies are identified starting from the different outgoing connectors (service, interface, extension, or package connectors) of this plugin element.

iv) **Block identification and feature naming**: We used the interdependent elements algorithm in the identification process. Once the blocks are identified they have names like block 0, block 1, etc. They are then automatically renamed

[2]For more details see: http://tiny.cc/wuwv7y

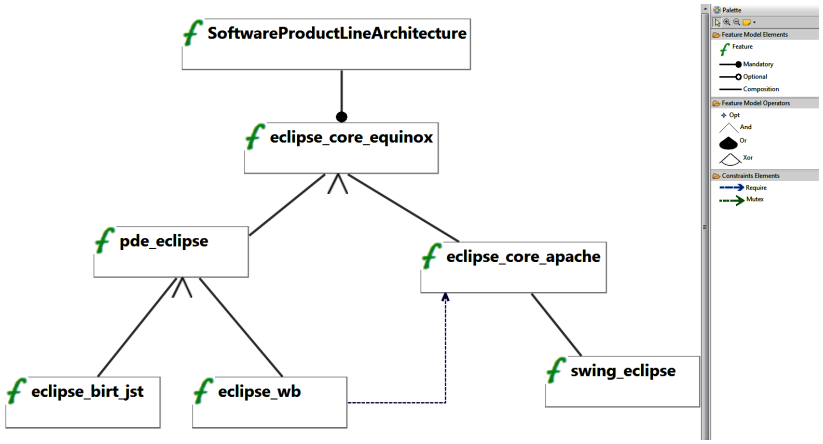

Fig. 4. Example of SAPL for three Eclipse Variants

using *VariClouds* [8], in order to give them more representative names.

D. Multi-views SAPL Construction for Eclipse SA Variants

The generation of the SAPL is implemented as a separate plug-in. 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 lines, such as applications built with Java 9+ module system. This plug-in 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. Figure 4 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 block "`eclipse core equinox`" is common to the three variants. The edge with a dashed line represents a discovered require dependency.

Besides, Figure 5 shows an excerpt of the architecture of the block "`eclipse birt jst`". As we can see, the component "`BIRT Emitter Conf. Plug-in`" provides an extension-point which is extended by several plugins. In this architecture, the set of provided / required elements that are not connected to others components, represent elements that are connected to components located in other features. Our tool enables the developers to merge two or several features in a single architecture to visualize the structural dependencies between them.

E. Architecture and Product Derivation

In order to create a new Eclipse variant, the developer can use the `featureIDE` tool for configuring manually the SAPL and select a set of desired features among the identified list. Before deriving the variant, we offer to the developer a way for mapping this configuration into an architecture model for this variant. This architecture model represents the structure of selected features and their relationships without variability information, which is useful for the understanding purpose. At the end, the new variant can be derived by collecting the extracted software assets which correspond to the selected features.

IV. Evaluation of the Process

In order to evaluate the performance of our approach, we have conducted a set of experiments on a set of Eclipse releases (product variants). The selected variants are the 12 Eclipse IDE Kepler SR2 releases[3]. The size of these variants varies from 110 to 323 MB and the number of components varies from 213 to 892 components.

In this experiment, we have addressed the following research questions:

- **RQ1**: What is the performance of our SAPL reverse engineering process?
- **RQ2**: What is the additional cost induced by the proposed Muti-Views SAPL construction?
- **RQ3**: What is the performance of block identification and feature naming?
- **RQ4**: What is the additional cost induced by the variants derivation using our approach?

A. Evaluation Protocol

a) : For answering **RQ1**, we have measured the accuracy of SAPL recovery process and the SA / product derivation. We have followed the following steps:

1) We have used our approach to recover the SAPL from the candidate variants. Next, we have used `FeatureIDE` Framework[4] for configuring a new variant in which we have selected `Xtext` feature and all the features that are identified from Eclipse Modeling variant.

[3]Downloaded from: https://bit.ly/2uylkT8

[4]FeatureIDE Framework: http://www.featureide.com/

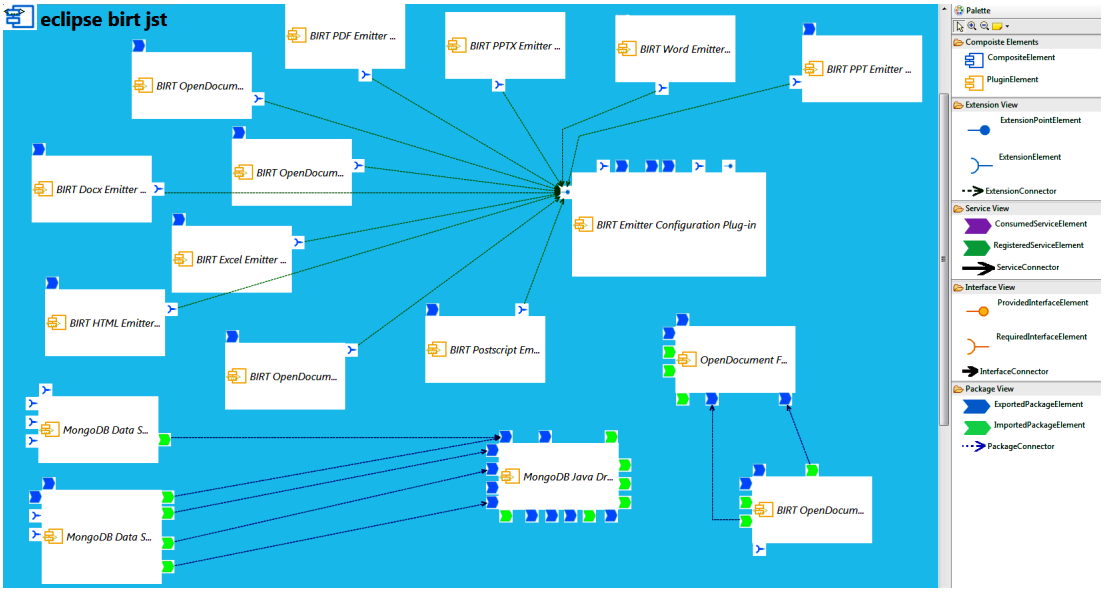

Fig. 5. Excerpt of the SA of "`eclipse birt jst`" Block (Extension and Package view)

2) After that, we have used our tool to derive the SA corresponding to this configuration.
3) Without our approach, we installed manually the `Xtext` in the Eclipse Modeling variant (click on Help->Install Modeling Components->set Xtext...).
4) At the end, we have used the *Architecture2Architecture* (`a2a`) [10] metric to measure the architectural change between the derived SA and the SA of the variant that is created manually (in step 3). The architectural change refers to the addition, removal, and modification of components and their elements.

$$a2a(A_i, A_j) = (1 - \frac{mto(A_i, A_j)}{aco(A_i) + aco(A_j)} * 100\% \quad (1)$$

where, mto(A_i, A_j) is the number of operations needed to transform architecture A_i into A_j and $aco(A_i)$ is the number of operations needed to create architecture A_i from a "null" architecture.

b) : For answering **RQ2**, we have used a set of measures for comparing the size and complexity of the recovered SAPL views using our process.

c) : For answering **RQ3**, we have measured the `precision` and `recall` on the results of block identification process. Indeed, we have compared the content of each identified block with the content of Eclipse features (obtained from "*features*" folders of the input variants). The evaluation is performed by comparing the plugins that belong to an identified block with plugins that are present in the "feature.xml" files of the common *features* folders of the artifacts that contain this block. To evaluate block naming, 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 [3]).

d) : For answering **RQ4**, we have compared the size and the number of components in "Block 0" (which is considered as the smallest variant that can be derived using our approach) with the size and number of components of the smallest input variant. This allows us to see to what extent is efficient the variant derivation using our approach.

B. Results and Discussion

*1) Experiment results for **RQ1**:* : We have used the `LoongFMR` implementation[5] of the `a2a` metric for comparing the architecture of the variant that is created manually with the architecture that is derived using our approach. The obtained value is `a2a` = **87%**. This value can be considered as a good result that indicates that the two architectures are not identical but almost the same. To understand why there is a little difference between them, we compared the new added (installed) plugins after the manual installation with the plugins in the architecture of the identified feature "`eclipse xtext`" using our approach. We observed that 56 new plugins (without counting plugins containing source code) are added to the variant after the manual installation of `Xtext` feature, but in the architecture of the identified feature ("`eclipse xtext`") using our approach, we only found 43 plugins. In fact, all these 43 plugins belong to the manually added plugins. By analyzing the remaining 13 plugins, either they represent plugins with older versions which already exist in the default variant, or they already exist but they are located in another identified feature where they must belong (for example, the plugin "*com.google.guava*" is located in the feature "`google common collect`"). This makes our variant more consistent than the variant that is created manually.

In addition, we have compared the size of the two variants in order to confirm that the derived variant using our approach

[5]Downloded from: https://github.com/csytang/LoongFMR

corresponds to the variant that is created manually. Indeed, the size of the Eclipse variant that is created manually is 256 MB, while the derived variant has a size of 268 MB. This minor difference in size is due to the fact that in the derived variant there are some plugins (such as *ch.qos.logback.slf4j*) that have been added for meeting the constraints that are discovered using But4Reuse. Examples of these constraints include: "`eclipse ecore` *implies* `eclipse Maven apache`" and "`Maven apache` *implies* `logback qos`", where the feature "`eclipse ecore`" is required for installing `Xtext`.

*2) Experiment results for **RQ2**:* In Table I the recovered SAPL views are compared in size with the whole SAPL (all views together). First, we can see that the number of elements (per block or per artifact) in each SAPL view is much less than the number of elements in the whole SAPL. This confirms our intuition that focusing on a single view allows to reduce the size and complexity of the SAPL. We can remark also that the number of elements in the extension view is less than the number of elements in the other views. This supports our idea that the developer needs to start with an architecture view that contains a few elements (only the plugins and their extensions). After that, (s)he can pass to another view with more information about other kinds of dependencies.

*3) Experiment results for **RQ3**:* We have found 971 "*features*" folders in all the chosen variants which is larger than the number of identified blocks. This is due to the fact that in our adapter, several features are merged into a single block. We depicted in Table II the obtained values of `precision` and `recall`. As we can see, we have obtained good values of recall and precision for `Block 0`. This means that, our tool can create an operational and minimal variant with an error rate almost equal to zero.

For the other blocks, we also obtained quite good scores, especially when we recover the SAPL with the Interface view. We can observe that the *median* scores are quite low compared to the `Block 0` scores, but, they are relatively good which illustrates the effectiveness of our approach. This decrease compared to `Block 0` is explained by the fact that there are some blocks which contain a few number of plugins (sometimes one or two plugins), and they are not present in the corresponding Eclipse features. Hence, we obtained low scores of precision and recall which decreases the *median* scores. But, we can observe from the scores that the number of these blocks is very low.

Besides, we present in Figure 6 the 62 identified blocks (for the interface view). The common part between all the variants (in blue color) represents the "Block 0" that is named "`eclipse core equinox`". This represents the core components that must exist in each variant. We note that all block names in this figure are assigned automatically. When comparing them with names given by the experts, more than 70% of names are the same, thanks to the word cloud that is used to name these blocks starting from words that are extracted from the elements names.

*4) Experiment results for **RQ4**:* After applying our SAPL reverse engineering process on the input variants, we have derived a new variant with only the "Block 0". We have found that the size and the number of components in this block is respectively 62.6 MB and 193 components. This is much less than the size and the number of components of the smallest input variant (IDE for Testers with 110 MB). This means that the minimalistic Eclipse variant (that contains the required minimum) that can be derived starting from the input variants can have only this size. This demonstrates the efficiency of our process. Instead of installing a default variant, we can configure an architecture, optimized according to developer requirements, and voluntarily minimalistic.

V. Related Work

Wesley et al. [6] presented a complete survey on the existing SPLE approaches. Three ways for adopting SPLE are exposed: (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. But4Reuse is a framework for extractive SPL adoption. Several extensions of BUT4Reuse have already been developed and published in [11], [12], [13]. Martinez et al. [11] 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. [13] proposed a SPL extraction approach from Bytecode based applications.

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 [14], [10], [15]). Most of these approaches are proposed for a single software architecture recovery. Lutellier et al. [10] present a comparative analysis of six SAR techniques. Maqbool et al. [15] 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 [16] and [17] are based on the definition of a correspondence model between the code elements and the architectural concepts. In [18], [19] a component is considered as a group of classes collaborating to provide a system function. Seriai et al. in [20] used FCA to perform the component interface identification. The authors in [21] recover BPMN models starting from service oriented systems that have been generated from web applications. Some works have been proposed to recover software architecture at run-time. For instance, [22] 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 [23] 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 size of the recovered architectures.

In our approach, we assume that the SAs of the product variants can already exist and considered as inputs for our

TABLE I
VALUES OF SIZE FOR THE RECOVERED SAPL VIEWS *vs* WHOLE SAPL

SAPL	# of Blocks	Mean of Elem. per Artifact	Median of Elem. per Artifact	Mean of Elem. per Block	Median of Elem. per Block
Extension	67	3144	2848	51	7
Package	67	7942	7557	208	30
Service	61	54861	12174	29	8
Interface	62	8492	7896	376	57
Whole SAPL	77	74439	30475	664	102

TABLE II
PRECISION AND RECALL CALCULATION FOR THE BLOCK IDENTIFICATION STEP

	Extension View		Package View		Service View		Interface View	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
Block 0	**0.99**	**0.83**	**0.99**	**0.83**	**0.99**	**0.83**	**0.99**	**0.83**
Median	0.76	0.68	0.76	0.68	0.91	0.75	0.94	0.69

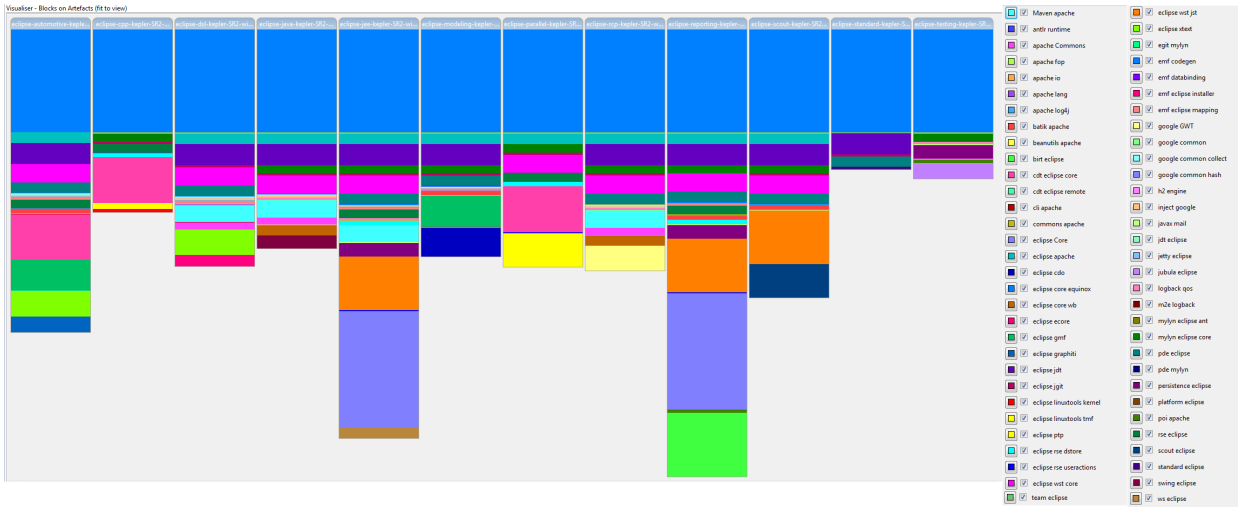

Fig. 6. Blocks per Variant: Interface View

SAPL-RE process. Otherwise, we can use one of the existing approaches for recovering them. However, the organization of features in the recovered SPLA is based on the result of the blocks identification and constraints 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 SPLA. [24] presented a mapping study of the existing SPLA recovery approaches. Shatnawi et al. [25] 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 [26] have proposed an approach for recovering SPLA from software 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. In addition, our inputs can be system variants or SA variants. The variability is identified starting from the elements in the input architectures. Wille et al. [27] 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 form the reconstructed SAPL. The proposed process is generic and can be applied for many component based-systems (or -software architectures).

VI. 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 product variants, these models become of greater interest since they enable also to choose the most appropriate variant. SPL Reverse Engineering (SPL-RE) processes enable to recover models with a better structure, since they factorize the variable part in the product variants and enable to see the variability points.

In our work we focused on component- and service-based systems and proposed in this paper: i) a (meta-)model for architectures of component/service-based software product lines, ii) the design of an adapter of a generic SPL-RE process (But4Reuse) for building architecture models (SAPL models) by analyzing product variants, iii) an implementation of this adapter specific to OSGi-based applications, and iv) an experimentation of this recovery process 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.

As perspectives to this work, we plan to study the enrichment of SPL reverse engineering of large component/service-based systems by including a learning module which exploits existing SPLs and their variants/features. 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 Java SE, EE, ME, TV, etc. From a tool-support point of view, we intend to enrich our implementation by capabilities such as software product configuration and derivation to complete the "loop".

Acknowledgement

The work of Tewfik Ziadi was supported by the ITEA3 15010 REVaMP2 project: FUI the Ile-de-France region and BPI in France.

References

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

[2] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich, "FeatureIDE: An extensible framework for feature-oriented software development," *Science of Computer Programming*, vol. 79, no. 0, 2014.

[3] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and 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.

[4] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Bottom-up technologies for reuse: automated extractive adoption of software product lines," in *Proc. of ICSE Companion*. IEEE Press, 2017.

[5] G. Perrouin, J. Klein, N. Guelfi, and J.-M. Jézéquel, "Reconciling automation and flexibility in product derivation," in *Proc of. the 12th SPLC*. IEEE, 2008.

[6] W. K. G. Assunção and S. R. Vergilio, "Feature location for software product line migration: a mapping study," in *18th SPLC, Companion Volume, Italy*, 2014.

[7] M. L. Kerdoudi, T. Ziadi, C. Tibermacine, and S. Sadou, "A bottom-up approach for reconstructing software architecture product lines," in *Proc. of the 13th ECSA: Companion Proceedings*. ACM, 2019.

[8] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Name suggestions during feature identification: The VariClouds approach," in *Proceedings of the 20th SPLC , Beijing, China*, 2016.

[9] J. McAffer, P. VanderLei, and S. Archer, *OSGi and Equinox: Creating highly modular Java systems*. Addison-Wesley Professional, 2010.

[10] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidović, and R. Kroeger, "Measuring the impact of code dependencies on software architecture recovery techniques," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 159–181, 2018.

[11] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and 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.

[12] L. Li, J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Mining families of Android applications for extractive SPL adoption," in *Proceedings of the 20th SPLC 2016, Beijing, China*, 2016.

[13] T. Ziadi and L. M. Hillah, "Software product line extraction from bytecode based applications," in *Proc. of the 23rd (ICECCS)*. IEEE, 2018, pp. 221–225.

[14] S. Ducasse and D. Pollet, "Software architecture reconstruction: A process-oriented taxonomy," *IEEE TSE*, vol. 35, no. 4, pp. 573–591, 2009.

[15] O. Maqbool and H. Babri, "Hierarchical clustering for software architecture recovery," *IEEE TSE*, vol. 33, no. 11, pp. 759–780, 2007.

[16] S. Chardigny, A. Seriai, M. Oussalah, and D. Tamzalit, "Extraction of component-based architecture from object-oriented systems," in *Proc. of WICSA*. IEEE, 2008.

[17] A. Seriai, S. Sadou, and H. A. Sahraoui, "Enactment of components extracted from an object- oriented application," in *Proc. ECSA*. Springer, 2014.

[18] S. Allier, H. A. Sahraoui, S. Sadou, and 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.

[19] S. Allier, S. Sadou, H. A. Sahraoui, and R. Fleurquin, "From object-oriented applications to component-oriented applications via component-oriented architecture," in *Proc. of the 9th WICSA, Colorado, USA*. IEEE, 2011.

[20] A. Seriai, S. Sadou, H. Sahraoui, and S. Hamza, "Deriving component interfaces after a restructuring of a legacy system," in *Proc. of WICSA*. IEEE, 2014.

[21] M. L. Kerdoudi, C. Tibermacine, and S. Sadou, "Opening web applications for third-party development: a service-oriented solution," *Journal of SOCA*, vol. 10, no. 4, pp. 437–463, 2016.

[22] G. Huang, H. Mei, and F.-Q. Yang, "Runtime recovery and manipulation of software architecture of component-based systems," *Journal of ASE*, vol. 13, no. 2, pp. 257–281, 2006.

[23] M. L. Kerdoudi, C. Tibermacine, and S. Sadou, "Spotlighting use case specific architectures," in *Proc. the 12th ECSA*. Springer, 2018, pp. 236–244.

[24] Z. T. Sinkala, M. Blom, and S. Herold, "A mapping study of software architecture recovery for software product lines," in *Companion Proceedings of ECSA*, 2018.

[25] A. Shatnawi, A.-D. Seriai, and H. Sahraoui, "Recovering software product line architecture of a family of object-oriented product variants," *J. Syst. Softw.*, vol. 131, no. C, pp. 325–346, Sep. 2017.

[26] H. Eyal-Salman and A.-D. Seriai, "Toward recovering component-based software product line architecture from object-oriented product variants," in *Proc. of SEKE*, 2016.

[27] D. Wille, K. Wehling, C. Seidl, M. Pluchator, and I. Schaefer, "Variability mining of technical architectures," in *Proceedings of the 21st SPLC - Volume A*. ACM, 2017.