Recovering Architectural Variability from Source Code

Crescencio Lima
Federal Institute of Bahia
Vitória da Conquista, Brazil
crescencio@gmail.com

Ivan Machado
Federal University of Bahia
Salvador, Brazil
ivan.machado@ufba.br

Matthias Galster
University of Canterbury
Christchurch, New Zealand
mgalster@ieee.org

Christina von Flach G. Chavez
Federal University of Bahia
Salvador, Brazil
flach@ufba.br

ABSTRACT

Context: Systematic variability management helps efficiently manage commonalities and differences in software systems (e.g., in software product lines and families). This enables the reuse of development artifacts in organizations and increases the quality of product variants. In software product lines, the product line architecture (PLA) is the core architecture for all product line variants. In practice, software architectures are often not documented in detail. Architecture recovery techniques can recover a system's architecture from development artifacts (e.g., source code). To recover the architecture of product lines, we need recovery techniques that are able to identify variability from different sources. *Goal:* We present SAVaR , an approach to recover architectural variability from the source code of product variants of a product line. SAVaR aims to help developers to (a) create architectural documentation for a product line, and (b) understand and improve the implementation of variability. SAVaR identifies the smallest subset of architectural information that is common across products of a product line. To limit the explosion of variability (and hence the complexity of architecture documentation) in the product line architecture , SAVaR allows architects to exclude architecture elements that appear in only a few product variants. *Method:* We performed an exploratory study with SAVaR to recover the architectures in ten academic product line projects. We verified how the elimination of exclusive optional modules improves the results of SAVaR . *Results:* The results showed that SAVaR is able to present improvements for the recovered PLAs and it helped to identify that some projects maintained the variability under control.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; *Software architectures*; *Empirical software validation*;

KEYWORDS

Software Product Lines; Product Line Architecture; Variability; Product Line Architecture Recovery

ACM Reference format:
Crescencio Lima, Ivan Machado, Matthias Galster, and Christina von Flach G. Chavez. 2020. Recovering Architectural Variability from Source Code. In *Proceedings of 34th Brazilian Symposium on Software Engineering, Natal, Brazil, 19–23 October, 2020 (SBES'20),* 10 pages.
https://doi.org/TBD

SBES'20, 19–23 October, 2020, Natal, Brazil

ACM ISBN TBD…$15.00
https://doi.org/TBD

1 INTRODUCTION

Variability is a characteristic of many software systems [10, 15] to manage commonalities and differences across software products, and to systematically accommodate reuse across organizations and product variants [11], for example, in software product lines, product platforms or ecosystems. Ideally, variability is identified and designed into products early, instead of discovered and addressed later in the life cycle [33]. Moreover, since variability is complex and multi-faceted, it impacts software development processes and practices (e.g., application and domain engineering practices in software product lines) as well as software development artifacts (e.g., software architectures and designs that enable variability). Hence, variability needs to be treated as a first-class citizen throughout software development [12].

In software product lines (SPL), one example of variability-intensive systems, variability is often supported during feature modeling or product configuration [2]. At the software architecture level, variability may be supported by a product line architecture (PLA) as a core architecture for all SPL products, representing the variation points and possible variants represented in a system's variability model [28]. The PLA provides a high-level description of mandatory, optional, and variable components in the SPL, including their relationships [14].

It is not uncommon for small and medium-sized companies to adopt the SPL paradigm using a *clone-and-own* approach, by copying, adding or removing functional features from existing products [29]. This approach leads to *ad-hoc* product portfolios of multiple yet similar product variants [9]. Furthermore, in this approach, a common PLA and its explicit description do not exist. With the growth of product portfolios, the management of variability and reuse becomes more complex [30]. However, a PLA for the SPL could be recovered from all the product variants and be used to understand the commonalities between the implementations of

products, variation points, etc. and eventually to guide systematic SPL evolution.

Software architectures can be recovered from source code or other available information (e.g., binaries) [7, 8]. However, software architecture recovery (SAR) for product lines is more challenging than recovering the architecture for "single" systems. This is because of the additional effort required to identify variability spread across multiple implemented product variants and the challenges related to representing variability at the architectural level. Existing work on PLA recovery [23] focused on the reuse of architectural information but lack information to support the architectural variability recovery, and in particular do not provide sound empirical evaluations.

In this paper, we present the Software Architecture Variability Recovery (SAVaR) approach for PLA recovery. SAVaR supports (a) the identification of variability-related information from source code, and (b) variability documentation at the architectural level. It comprises two techniques for automating the PLA recovery , and a guideline for how to use the techniques.

To evaluate SAVaR, we performed an exploratory study with ten academic SPL projects. The results showed that SAVaR is able to: (i) help to identify that the projects presenting small number of threshold values considered the variability upfront and (ii) present improvements for the recovered PLAs by reducing the number of optional modules.

The remainder of this paper is organized as follows. Section 2 discusses background work related to variability in software architecture, architecture recovery and product line architecture. Section 3 describes SAVaR, details of the two recovery techniques and the guideline for supporting the PLA recovery with SAVaR. Section 4 presents an exploratory study to investigate the application of SAVaR. Section 5 presents related work. Section 6 concludes the paper and summarizes its contributions and future work.

2 BACKGROUND

This section provides background information about variability in software architecture (Section 2.1) and software architecture recovery (Section 2.2).

2.1 Variability in Software Architecture

Variability is a concern of software systems that are developed for different deployment and usage scenarios. Variability is also reflected in the architecture of those systems [11, 15]. One type of variability-intensive systems are software product lines. Variability can therefore be represented in and assessed through a (documented) software architecture (for example, variability are layered or modularized component and connector models that contain constant elements and encapsulate parts of the system that may change [11]). [1]

To address variability at the architecture level in software product lines, and addressing [6, 10–12], the software engineering community introduced the concept of product line architectures (PLA) [1, 28]. PLA capture the core design of all products including variability and commonalities of several products [36]. Product line architectures are expected to enable reuse of existing structures with only a limited number of exchangeable components to tailor the system to each customer's specific needs [3].

Software architecture documentation may comprise one or more architectural views, such as a logical, physical, deployment or development view [18]. This provides different types of stakeholders who have different concerns with useful information [7].

Clements and colleagues argue that variability in an architecture for a family of systems or product line needs to document the variation points (places in the architecture where variation can occur, and associated alternatives), and the elements affected by the possible alternatives [7]. Moreover, if there are many variation points to be documented, a specific view that shows just the variation points can be used [7]. Therefore, variability can be presented in different architectural views.

2.2 Software Architecture Recovery

Software architecture recovery (SAR) comprises the techniques and processes to uncover a system's architecture from development artifacts (e.g., source code, binaries) [17]. SAR makes existing (but potentially undocumented) software architectures explicit, by providing missing architectural specifications or supporting the update of existing architecture documentation.

SAR requires different techniques and artifacts [32] to recover architecture-relevant information to be presented in different types of architectural views. For instance, the recovery of logical views [18] relies on the analysis of requirements models, use cases and activity diagrams. In this example, a top-down recovery process could start with high-level knowledge (requirements-related information) and discovers the architecture [8]. On the other hand, the recovery of physical and deployment views [18] requires the analysis of dynamic information such as runtime logs, network traffic, configuration files, and installation scripts. Similarly, the recovery of development views [18] follows bottom-up processes and static analysis to extract information from low-level artifacts (e.g., source code).

SAR for SPL is more complex than for single systems, because of the additional effort required to identify variability and commonalities from different product variants. For instance, the recovery of development views may require the analysis of `#ifdef` directives and the source code of several implemented product variants. Additional concerns when recovering product line architectures are how to abstract and represent variability and commonality in a single PLA, and how to present them in architectural views for different stakeholders. In this paper, we focus on the recovery of development views [18]. The development view represents systems components as collections of source code artifacts (e.g., files, functions, types) and relations between these system components (e.g., calls, uses, sets).

We highlight that the implementation of a specific function can differ from one variant to the other. However, this variability occurs at lower levels (e.g., two methods that implement different algorithms). Our objective is to identify the variability in a higher level of abstraction, i.e., at the architecture level. In this context,

[1] Note that in this work, we use the terms "architectural variability" and "variability in software architecture" interchangeably.

the lower level variability (e.g., inside methods) does not affect the structure of the PLA.

3 THE SAVAR APPROACH

This section introduces SAVaR, our approach for PLA recovery. SAVaR is semi-automatic and follows a *bottom-up* architecture recovery process. SAVaR supports a recovery workflow based on a sequence of *extract-abstract-present* activities [35]. Therefore, we first describe this sequence of *extract-abstract-present* activities (Section 3.1). Then, we discuss two techniques for automating the PLA recovery (Section 3.2) and a guideline to help practitioners use SAVaR (Section 3.3).

3.1 SAVaR Activities

Figure 1 presents SAVaR activities using the *Software and Systems Process Engineering Metamodel (SPEM)*.[2] We used SPEM to summarize SAVaR activities sequence, inputs, outputs, and roles in a high abstraction view. Adapting the roles involved in architecture recovery proposed by Garcia et al. [13] to a product line context, the following roles are involved: The *Recoverer* is responsible for executing the steps described in the recovery guideline (see Section 3.3). The *SPL Architect* is responsible for understanding and verifying the conformance of the recovered PLA. The *SPL Developer* is responsible for verifying the recovered architecture information based on the SPL implementation and checking the relationship between the SPL source code and the recovered PLA.

The recovered PLA is described in accordance with a reference metamodel [21, 26] as a comprehensive conceptual basis for variability at the architectural level. Next, we describe the SAVaR activities by following the flow presented in Figure 1 (the numbers of sections where we discuss each activity are also included in Figure 1). We discuss SPL information collection (Section 3.1.1), information extraction (Section 3.1.2), PLA recovery and variability identification (Section 3.1.3), and PLA presentation (Section 3.1.4).

3.1.1 SPL Information Collection. This activity receives the SPL source code and the guideline to support SAVaR as inputs. During the SPL information collection, the recoverer gathers the information about the SPL project by analyzing the source code and identifying the mechanism used to implement variability. Such information allows the generation of the products that will be used as input for the next activity.

Moreover, other assets such as feature model, documentation, domain knowledge, requirements, etc., provide additional support when they are available. However, to execute the PLA recovery with SAVaR only the source code (or binary) is mandatory; other information sources are optional. The output of this activity is a set of variants source code.

3.1.2 Information Extraction. In this activity, SAVaR extracts structured information based on the variants source code (semi-structured information). Several existing analysis and extraction tools that work for single systems can be used to extract information (i.e., source code models) required for SAVaR from individual product variants.

These tools extract low-level source code models for products implemented in one or more programming languages, for instance, Stan4J[3] (Java), Struct101[4] (Java), Understand[5] (C and Java), Analizo[6] (Java, C, C++, etc.), and PlantUML Dependency[7] (Java). cppstats[8], a toolsuite for analyzing cpp-preprocessor-based SPL, can be used to extract information from products with variability implemented by means of `#ifdef` directives.

The reuse of extraction tools may require the implementation of adapters to deal with different input/output formats. The selection of an extraction tool is based on the programming language and/or the mechanism used to implement the variability.

3.1.3 PLA recovery and Variability Identification. This activity receives the source code models provided by the information extraction activity. The models serve as input for the two recovery techniques presented later in Section 3.2. The techniques are used to identify (i) mandatory modules and relationships, that is, those modules that are present in all the variants, and (ii) optional modules and relationships that are present in only some variants and therefore represent variability between product variants.

Modules can be packages, classes, methods, functions, and other units of modularization. After identifying the variability, the variability-aware architectural models that comprise a PLA can be generated. In our approach, the architectural models conform to the specification of a variability-aware architectural metamodel [21, 26, 33], that includes modeling elements for representing mandatory and optional modules and their relationships.

3.1.4 Presentation of the Recovered PLA. The recovered PLA can be presented using visual notations for development views (e.g., UML class diagrams, module dependency graphs or design structure matrices). In SAVaR, these diagrams are enhanced to explicitly provide information about architectural variability and related assets. As mentioned above, the recovered architecture conforms to a variability-aware architectural metamodel.

Figure 2 presents the metamodel to support SAVaR. We simplified the metamodel proposed by Thiel and Hein [34] and updated it according to the ISO/IEC/IEEE 42010 [16], a standard for software architecture descriptions. Moreover, we included an adaptation of the *Architecture Variability Extension* and *Design Element Extension*.

In the metamodel, a *PLA Description* is an architecture description enriched with explicit information about the *Architectural Variability Model* of the software product line *SPL-of-interest*. This means that PLA architecture models must be concerned with the representation of *Architectural Variation Points*. The architectural variability models consist of *Architectural Variability* that is represented by different *Architecture Views*.

The variability information is represented in development views and architectural elements such as packages and classes. We used the *Variation Point Specification* for describing the *Optional Module* (for elements that are implemented by only some variants) and the

[2] http://omg.org/spec/SPEM/2.0/

[3] http://stan4j.com/
[4] http://structure101.com
[5] http://scitools.com/
[6] http://www.analizo.org
[7] http://plantuml-depend.sourceforge.net
[8] http://fosd.net/cppstats

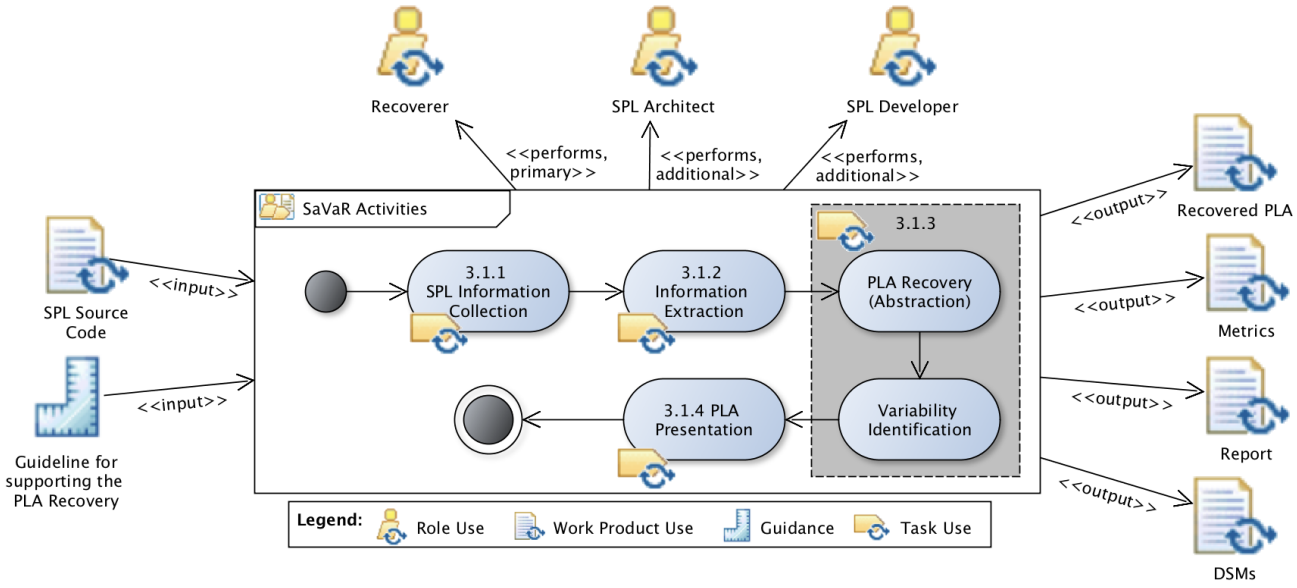

Figure 1: SAVaR Activities

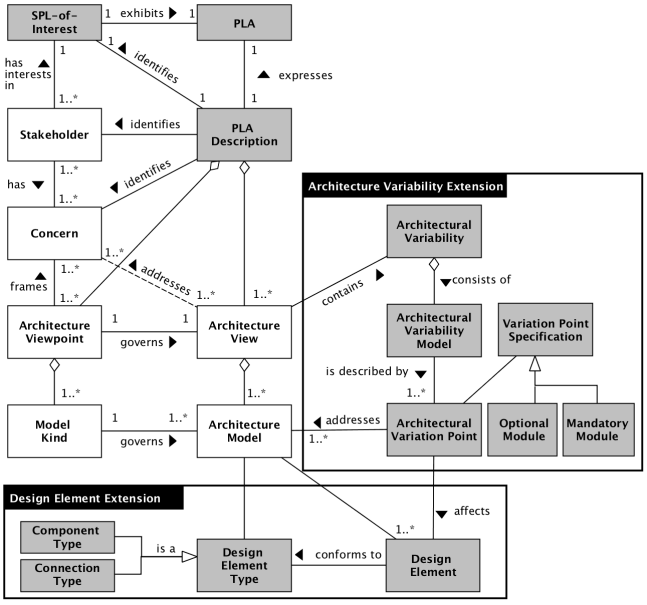

Figure 2: Conceptual elements of SPL and SA based on ISO/IEC/IEEE 42010 (adapted from [34])

Mandatory Module (for elements that are implemented by all the variants).

3.2 Techniques Implemented in SAVaR

We developed one technique to address the identification of architectural variability for a given SPL from source code and another technique to filter the recovered variability information. The first technique relies on the extracted architectures of variants (*Extract-and-Merge*). It provides high-level source code models annotated with information about variability. Details are discussed in Section 3.2.1. The second technique improves the results of the Extract-and-Merge. It implements a threshold analysis to reduce the "amount" of variability information represented in the recovered PLA by focusing on ignoring modules that were implemented by few variants. Details are in Section 3.2.2.

3.2.1 Extract-and-Merge Technique. This technique performs variability recovery based on merging the architectures extracted from products. It uses extraction tools to recover a set of architectures, one for each product. The recovered architecture for each product variant is represented as a set of modules and their relationships. Pattern matching based on module name is used to identify mandatory and optional modules from the extracted architectures of the analyzed product variants.

We assume that product portfolios preserve the name of mandatory modules in the implementation of different product variants. A module (e.g., a class in Java-based systems) that is present in every recovered product architecture is labelled as a "mandatory module" while modules that appear in one or some of the product variants are labelled as "optional module". Relationships between mandatory modules are labelled as "mandatory relationship". The set of mandatory modules and relationships, and optional modules and relationships defines the recovered PLA for the SPL and its products [23].

Figure 3 illustrates the technique. The core of the technique is a merging algorithm ② to identify variability. The input into the algorithm are the architectures extracted from the products ①. The technique identifies and groups mandatory modules (e.g., *array_1:* classes A and B) and the mandatory relationship (e.g., *array_3*), and organizes optional modules (e.g., *array_2:* classes C, D, and E) and

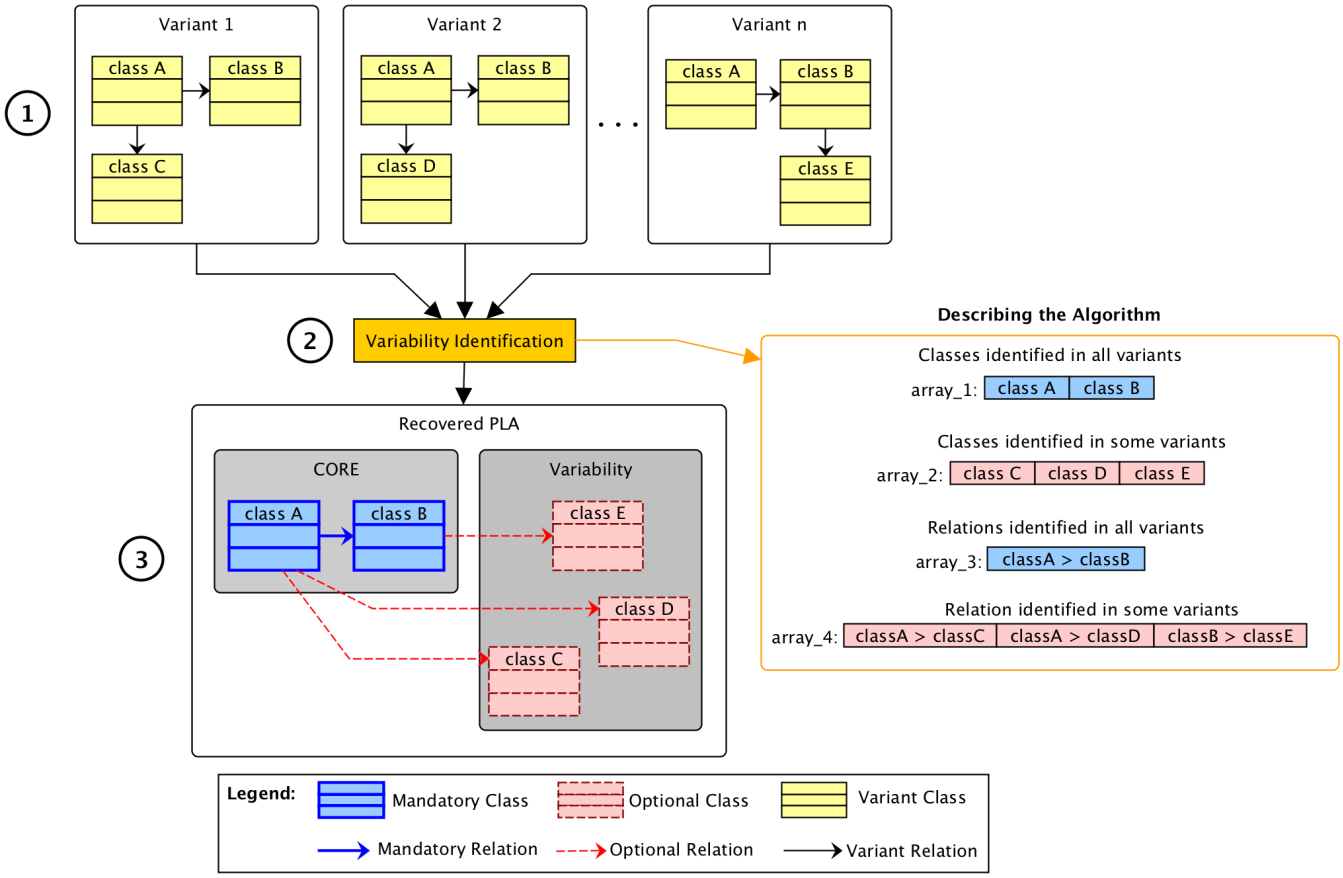

Figure 3: PLA recovery based on the architecture of variants - *Extract-and-Merge technique*

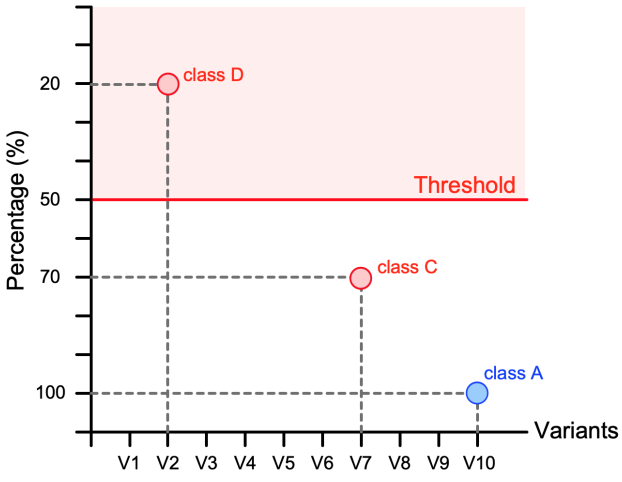

Figure 4: Example of threshold technique

optional relationships (e.g., *array_4*). The output is the recovered PLA (③ in Figure 3).

3.2.2 Threshold Technique. In our previous work [19, 20], We developed a threshold analysis technique that identifies *exclusive modules*, i.e., optional modules that appear only in a small number of product variants, and whose inclusion in the visual presentation of the recovered PLA would result in a cluttered architecture. Therefore, the technique suggests exclusive modules (and relationships) as candidates to be removed from the PLA recovery process. The decision on whether or not to remove a module is based on a threshold that captures the minimum number/percentage of product variants a module appears in. Only including modules that exceed that threshold keeps potential variability explosion under control, by excluding exclusive modules and only keeping modules implemented in the majority of the products.

To determine the threshold, we rely on the number of occurrences of modules (e.g., package, class, or relationship) in the products. The technique prioritizes the mandatory (core) modules and the modules implemented in most variants.

Figure 4 illustrates how the threshold technique works. In this example, SAVaR considered ten variants for the PLA recovery. Class A was implemented by all the ten variants (100%), class C was implemented by seven variants (70%), and class D was implemented by only two variants (20%). We set up the threshold to 50%. Therefore,

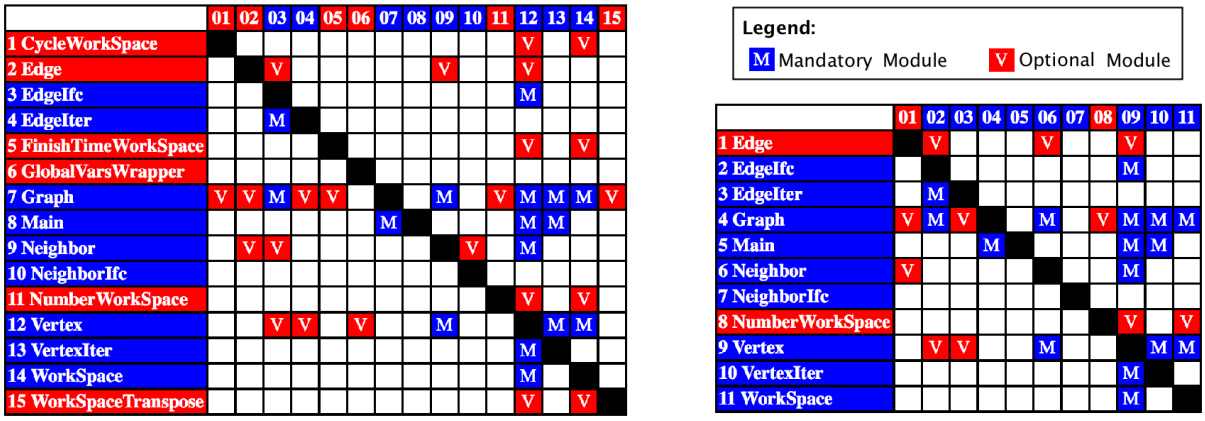

Figure 5: Applying threshold technique on the GPL SPL project (a) DSM - threshold: 0% and (b) DSM - threshold: 47%

classes A and C were kept in the product line architecture, while class D was not considered as input for recovering the PLA.

Figure 5 presents an example output of the threshold technique. We used the Design Structure Matrix (DSM) generated by SAVaR to visualize the output. In this example, we applied the technique on the Graph Product Line (GPL) project[9] which is an SPL for implementing graph manipulation libraries. By analysing the Inspection Report provided by SAVaR we identified the threshold values and extracted four PLAs based on the recovery of the variants modules.

In Figure 5, we show the first PLA recovery (without threshold applied) and with a threshold of 47% (i.e., modules need to appear in at least 47% of the variants), it reduced the recovered information (i.e., optional modules and their relationships) by 30%. The technique kept the mandatory modules and the optional modules that are implemented in the majority of the variants (e.g., Edge and NumberWorkSpace).

3.3 Guideline for Supporting PLA Recovery

In our previous work [23], we identified the lack of guidelines to support PLA recovery. Therefore, we documented a guideline to help practitioners use SAVaR. We performed a set of exploratory studies [5, 19, 20, 22, 23] that, as one finding, allowed us to define the guideline. The guideline describes a realistic PLA recovery scenario and supports identifying variability at the architectural level, by providing steps, a recovery problem definition, technique recommendation, hints, and potential pitfalls.

Figure 6 presents the *Generate Variants* guideline that helps recover PLA using variants (i.e., SPL products). Due to space limitation, we only provide a summary of the guideline. The complete guideline including its detailed description can be found at: http://sbes2020.herokuapp.com/.

[9] https://bit.ly/2B9MtSU

4 EMPIRICAL EVALUATION

This section describes how we evaluated SAVaR. We followed the Goal-Question-Metric (GQM) method to define the scope of the evaluation [4].

Evaluation goal: *evaluate how* SAVaR *and the guideline support the cost-effective PLA recovery through the identification and removal of exclusive optional modules.*

Based on the evaluation goal, we defined a research question: *To what extent can the elimination of exclusive optional modules improve the results of* SAVaR *?*

This research question focuses on the quality of the recovered PLA when removing exclusive modules (i.e., optional modules that appear in a small percentage of variants). We evaluate the quality of such PLA quantitatively based on the metrics introduced next.

4.1 Metrics

Product line architecture metrics support the evaluation of variability-aware software architectures. In our work, we use two sets of metrics to evaluate the structure of the recovered PLAs, their components and relationships: the metrics proposed by Zhang et al. [37] and the metrics proposed by Oliveira-Junior et al. [27].

Zhang et al. [37] proposed a set of metrics for assessing the quality of a PLA based on the similarity and variability of its components and relations. The "Structure Similarity Coefficient" (SSC) measures the structure similarity of PLA components. Similarly, the "Structure Variability Coefficient" (SVC) measures the structure variability of PLA components. Given C_c (the number of common components in the PLA) and C_v (the number of variable components), SSC and SVC are defined as follows [37]:

$$\text{SSC} = \frac{|C_c|}{|C_c|+|C_v|} \qquad \text{SVC} = \frac{|C_v|}{|C_c|+|C_v|}$$

The "Relation Similarity Coefficient" (RSC) measures the similarity of relations between components in a PLA and the "Relation Variability Coefficient" (RVC) measures the variability in relations of components in a PLA. Given R_c (the number of common relations in the PLA) and R_v (the number of variable relations), RSC and RVC are defined as follows [37]:

Generate Variants
Intent: Recover product line architecture from a set of generated variants.

Problem
Product line architecture is not explicitly documented; only implementation artifacts are available One alternative is to recover the PLA based on the source code of the product line. If variants (in this case SPL products) are generated by tools such as FeatureIDE [25], configuration of variants and variant generation are necessary to populate the set of variants.

Recommended technique.
Generate variants that represent different configurations and use them as input to a technique that combines an existing set of variants to recover the PLA.

Steps.
(1) Obtain the variants' source code;
(2) Select an extraction tool to recover the variants' structural information;
(3) Perform the variability identification using tool support to automate the process;
(4) Analyze PLA metrics (SSC and SVC, RSC and RVC);
(5) Analyze the other outputs provided by SAVaR (report and visualization of the recovered PLA).

Hints for improving the PLA recovery.
- Use the report to identify the modules' frequency;
- Perform threshold analysis to identify possible improvements in the recovered PLA;
- FeatureIDE variant generator provides automation to define all combination of products and a T-wise algorithm that creates a set of relevant variants that can be used in projects with a high number of variants.

Potential pitfalls.
- When the variants source code is not available, a variant generator tool is needed to instantiate the products;
- Extracting all variants in projects with a high number of optional features could be cumbersome;
- One alternative to the threshold technique is to manually select a number of relevant variants that can impact the accuracy of the recovered PLA.

Figure 6: *Generate Variants* guideline.

$$\text{RSC} = \frac{|R_c|}{|R_c|+|R_v|} \qquad \text{RVC} = \frac{|R_v|}{|R_c|+|R_v|}$$

The SSC and SVC metrics are highly related given that the sum of SSC and SVC will be always 1. Values close to 1 for SSC means that there are few optional components, and values close to 0 means that the PLA of the different variants does not have many components in common. The RSC and RVC metrics behave similar to SSC and SVC, i.e., their sum will always be 1. Values of RSC close to 1 mean that there are few optional relations, while values close to 0 mean that the relations in the architectures of different variants do not have many relations in common.

Oliveira-Junior et al. [27] proposed a metrics suite for PLA evaluation based on UML models: (i) ClassOptional (CO) counts the number of classes implemented by a subset of variants; (ii) OptionalRelation (OR) counts the number of relationships between classes implemented by a subset of variants; (iii) ClassMandatory (CM) counts the number of classes implemented by all the variants; and (iv) MandatoryRelation (MR) counts the number relationships between classes that are implemented by all the variants.

We searched the literature to identify metrics in the context of PLA recovery. The metrics mentioned in this Section were chosen because they were the most suitable metrics for evaluating development views as created by SAVaR.

Table 1: Analyzed Projects

Projects	#V	#C	avg	#E	#R	#P	Gen.
Desktop S.	462	18942	41	12504	30126	10	AH
GOL	64	1344	21	1197	1998	4	NA
GPL	156	2340	15	1843	4341	7	CD
Health W.	10	1396	136	1113	4857	11	NA
MobileM.	8	346	43	243	406	8	NA
Prop4j	452	6328	14	3648	6710	10	FH
Message	10	680	68	493	743	5	Ant
VOD	32	1344	42	1184	2082	3	NA
Webstore	10	710	71	534	1408	4	Ant
ZipME	31	992	32	897	1226	4	NA
Total	1226	33783	483	23746	53897	66	-

Legend: [**#V**] Number of variants, [**#C**] Total number of classes analyzed, [**avg**] average number of classes per project, [**#E**] Total number of modules analyzed, [**#R**] Total number of relations analyzed, [**#P**] Number of execution of SAVaR, [**Gen.**] Variants generation, [**AH**] AHEAD, [**NA**] Not Available, [**CD**] CIDE, [**FH**] FeatureHouse

4.2 Projects

Table 1 presents descriptive data from the ten open source projects selected for our empirical evaluation [10]. We selected projects with different number of variants (**#V**), number of classes (**#C**), average number of classes per project (**avg**), and strategies for variants generation (**Gen.**). There are different composers for feature-oriented programming such as FeatureHouse, AHEAD, CIDE, etc. For this reason, we used different types of variants generation and projects to verify if SAVaR support them.

We used the FeatureIDE [25] to generate the variants from AHEAD, CIDE, and FeatureHouse composers. Projects implemented with `#ifdefs` used Ant build for variants generation. We used the other projects (**NA** - Not Available) variants source code available in the projects' repositories.

[10] https://bit.ly/2BcQQwg

4.3 Preparation

We collected information about the projects and downloaded the source code and other assets from their repository. We identified the mechanism used to implement the variability because the selection of the recovery techniques and extraction tools depends on them (Section 3.2).

Then, we extracted each variant structural information. We performed the variability identification. We mapped the mandatory modules that were implemented in all the variants and the optional modules that were implemented in only some variants.

With the recovered PLA outputs, we analyzed the metrics and reports. These outputs were used to suggest improvements to the results. We collected the modules implementation frequency to define the threshold values.

Based on the threshold, we performed the PLA recovery again. In this way, we provided a set of recovered PLAs allowing architects and developers to select the PLA according to their interests.

4.4 Analysis and Interpretation

Table 2 presents the collected metrics for PLA recovery analysis within the threshold results. We executed SAVaR according to threshold values based on a report generated by the approach. The report identifies the modules according to their existence in the variants. For instance, when a class is implemented in all the variants, the report informs that this class appears in the implementation of 100% of the variants.

Due to space limitation, we omitted some threshold results. For instance, we executed the PLA recovery 10 times for the Desktop Searcher project according to the threshold values, but Table 2 only shows seven results. The complete experimental setting and results can be found at the paper website[11].

4.5 Results

To answer the research question, we execute the SAVaR according to each project threshold values. Then, we compared the results to identify suggestions of improvements on the recovered PLAs. The implementation of threshold technique allowed the reduction of the number of optional classes without eliminating variants. It is an alternative to the solution we proposed in our previous study [19, 22]. Instead of identifying and eliminating outliers (variant that introduces a high number of optional modules that are implemented in only that variant), we kept all the variants during our analysis.

Projects such as GOL, VOD, Webstore, and ZipME allowed a small number of executions of SAVaR because the SSC and SVC values were balanced. Such balance may indicate that these projects considered the impact of variability on the architecture upfront during the development phase.

Moreover, the threshold technique allowed us to improve some projects metrics such as for MobileMedia. In this case, the majority of the variants implemented some classes such as AlbumData (87%), AddPhotoAlbum (76%), and ImageAccessor (75%). In other words, during the evolution of the SPL, stakeholders should consider making these features and these classes *mandatory*.

[11] https://sbes2020.herokuapp.com

Table 2: Recovered Metrics from the PLAs

TH	SSC	SVC	RSC	RVC	CO	OR	CM	MR	%rd
Desktop Searcher									
00%	0.27	0.73	0.09	0.91	30	134	11	14	-
04%	0.29	0.71	0.11	0.89	28	118	11	14	10%
25%	0.30	0.70	0.16	0.84	25	72	11	14	36%
29%	0.32	0.68	0.18	0.82	24	66	11	14	40%
37%	0.33	0.67	0.19	0.81	22	58	11	14	45%
43%	0.35	0.65	0.20	0.80	21	55	11	14	47%
49%	0.52	0.43	0.38	0.62	10	23	11	14	70%
GOL									
00%	0.62	0.38	0.69	0.31	8	11	13	24	-
50%	0.76	0.24	0.80	0.20	4	6	13	24	17%
GPL									
00%	0.60	0.40	0.41	0.59	6	23	9	16	-
16%	0.64	0.36	0.43	0.57	5	22	9	16	04%
39%	0.69	0.31	0.48	0.52	4	17	9	16	15%
47%	0.80	0.20	0.59	0.41	2	11	9	16	30%
Health Watcher									
00%	0.42	0.58	0.29	0.71	91	550	66	233	-
11%	0.49	0.51	0.32	0.68	67	501	66	233	08%
21%	0.54	0.46	0.38	0.62	55	367	66	233	24%
31%	0.56	0.44	0.40	0.60	51	360	66	233	25%
41%	0.58	0.42	0.51	0.49	46	230	66	233	39%
MobileMedia									
00%	0.12	0.88	0.02	0.98	52	148	7	3	-
13%	0.15	0.85	0.03	0.97	41	94	7	3	31%
26%	0.20	0.80	0.05	0.95	29	54	7	3	56%
38%	0.25	0.75	0.08	0.92	22	39	7	3	67%
Prop4j									
00%	0.07	0.93	0.00	1.00	13	67	1	0	-
01%	0.08	0.92	0.00	1.00	12	49	1	0	24%
42%	0.09	0.91	0.00	1.00	11	13	1	0	70%
48%	0.10	0.90	0.00	1.00	10	12	1	0	72%
Message									
00%	0.63	0.37	0.57	0.43	22	40	38	54	-
41%	0.84	0.16	0.80	0.20	7	12	38	54	28%
VOD									
00%	0.76	0.24	0.70	0.30	10	23	32	55	-
Webstore									
00%	0.71	0.29	0.68	0.32	20	57	48	122	-
21%	0.96	0.04	0.91	0.09	2	11	48	122	26%
ZipME									
00%	0.80	0.20	0.69	0.31	6	14	25	32	-
51%	0.96	0.04	0.94	0.06	1	2	25	32	22%

Legend: **[TH]** Threshold, **[SSC]** Structure Similarity Coefficient, **[SVC]** Structure Variability Coefficient, **[RSC]** Relation Similarity Coefficient, **[RVC]** Relation Variability Coefficient, **[CO]** Class Optional, **[OR]** Optional Relation, **[CM]** Class Mandatory, **[MR]** Mandatory Relation, **[%rd]** Percentage of reduced information per threshold cut

Projects with high value of SVC and RSC (close to 1) could suffer from variability explosion. For instance, Prop4j allows the creation of 4,100 variants. During evolution, it is hard to maintain and propagate any changes. In order to reduce variability explosion, the threshold technique was able to improve the results slightly even with a 70% in information reduction. Our report identified that the majority of the variants implemented two classes: Literal (99%) and SatSolver (98%). By considering these classes as mandatory, the values for these metrics improved.

The metrics (high value of SSC and RSC) indicate that GOL, VOD, Webstore, ZipME, GPL and Message variability can be improved. On the other hand, projects with a high value of SVC and RSC (e.g., Desktop Searcher, MobileMedia, and Prop4j) indicate that improvements in the definition of mandatory modules are necessary.

We found that the information reduction provided by the threshold technique allowed to balance SSC and RSC, and RSC and RVC. It is relevant to support and raise the abstraction level at the architectural level. Moreover, in some cases, the technique reduced the information up to 70% and provided a balance in metrics values.

4.6 Threats to Validity

The following threats to validity potentially interference with our evaluation.

4.6.1 Internal Validity. We identified a selection effect during the selection of the variants. In some cases, when we considered all the variants in the recovery, the PLA was composed by only optional classes. To reduce the noise in the representation, we implemented the threshold technique.

The theory is not clear enough regarding the effectiveness of PLA recovery and improvement of the recovered PLA. We based our analysis on metrics and PLA representation. Even though, we cannot reject stakeholders' influence.

4.6.2 External Validity. The selected projects represent a small portion of possible projects. However, the study proves another case that can help build evidence regarding the impact of variability in the context of PLA recovery. Another issue we found is related to classes implementing the same logic but, using different names. We eliminated information specific to projects to reduce this issue and selected variants in projects developed by the same team.

We only considered open source projects. It may under-represent the SPL domain and thus not give the full picture of the problem The projects evolved over the years and new ideas were included contributing for the maturity and raise of the complexity of the projects.

The main threat to this study was the sample size. From the 1226 variants, we focused on the classes and their relationships. SAVaR also supports packages and files abstraction. However, since the purpose of this study was to provide evidence on how the threshold for including or excluding modules in the PLA improves the effectiveness of the PLA recovery, we understand that for generalizing such findings we need a larger sample.

5 RELATED WORK

Our work focuses on the recovery of architectural variability. We are concerned with the bottom-up recovery of a PLA from source code that captures the variability and commonalities of several related variants.

Few works address the recovery of architectural variability to capture a PLA. We previously performed a systematic mapping study to understand the relationship between PLA and SAR and to characterize how existing research supports PLA recovery, and to identify research trends and gaps [23]. The majority of research addressed some aspects of SPL such as reuse, variability, etc., but lacked proper empirical evaluation or detailed information to support PLA recovery and to apply proposed recovery techniques. Also, just a few works focused on the recovery of variability at the architectural level [30, 31].

Shatnawi et al. [30, 31] addressed PLA recovery with by comparing components (classes and interfaces) recovered from different versions of the same SPL. The authors relied on Formal Concept Analysis (FCA) to analyze variability and create a variability model. In our study, we used several variants as input to the PLA recovery process. For each SPL product, we extracted structural information from source code and collected information about variability found within classes, packages, and their relationships.

Linsbauer et al. [24] presented an approach for extracting structural information from related product variants to recover a feature model for the SPL. Likewise, our approach supports the extraction of structural information from the source code of related variants, but with the goal of recovering a PLA for the SPL.

Related approaches do not support recovering architectural variability information from source code and there is a gap regarding variability identification from variants that are implemented using a clone-and-own strategy.

6 CONCLUSIONS

In SPL, the PLA provides information about the common and variable architectural elements. Since not all projects have their PLA documented, PLA recovery provides support for stakeholders during maintenance and evolution.

In summary, this paper contributes the following:

(1) We developed SAVaR an *approach for PLA recovery from the source code of variants*; the approach includes a guideline to aid the recovery process. SAVaR supports the link between the SPL source code and its architecture to maintain them in sync.
(2) We implemented *a technique for automating the variability identification at the architectural level.* The recovered variability information from SPL projects, together with its representation at the architectural level provide up-to-date structural PLA documentation, synchronized with SPL source code, that can be useful for SPL stakeholders to perform their tasks.
(3) We proposed and implemented a *threshold technique to tame variability explosion and improve the recovered PLA results* The technique leverages the reduction of the variability in the recovered PLA while keeping all available variants in the analysis. It provides a set of outputs that can be selected according to stakeholders' interests. The threshold technique maintains the core elements (classes

and its relations) implemented by the majority of the variants and therefore emphasizes the information relevant to most variants but ignores specific information from some variants.

(4) We evaluated SAVaR with the application of the threshold technique in ten open source software systems. We followed the Generate Variants guideline for each threshold value and we performed a comparison among the recovered PLAs to identify improvements in SAVaR results.

As future work, we intend to create a decision model based on SAVaR executions. Also, we intend to extend the evaluation studies to include highly configurable systems using SAVaR . Other research opportunity is to introduce search-based software engineering to automate the threshold values identification.

ACKNOWLEDGMENTS.

This research was partially funded by INES 2.0, CNPq grant 465614/2014-0, and FAPESB grants JCB0060/2016 and BOL2443/2016. We would like to thank the reviewers and Jabier Martinez for his thoughtful comments.

REFERENCES

[1] Faheem Ahmed and Luiz Fernando Capretz. 2008. The Software Product Line Architecture: An Empirical Investigation of Key Process Activities. *Inf. Softw. Technol.* 50, 11 (2008), 1098–1113.

[2] Sven Apel, Don Batory, Christian Kastner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer.

[3] Jan Bosch. 2000. *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

[4] VRBG Caldiera and Dieter Rombach. 1994. The goal question metric approach. *Encyclopedia of software engineering* 2, 1994 (1994), 528–532.

[5] Mateus Passos Soares Cardoso, Crescencio Lima, Eduardo Santana de Almeida, Ivan do Carmo Machado, and Christina von Flach G. Chavez. 2017. Investigating the Variability Impact on the Recovery of Software Product Line Architectures: An Exploratory Study. In *Proceedings of the 11th Brazilian Symposium on Software Components, Architectures, and Reuse.* ACM, 12:1–12:10.

[6] Lianping Chen and Muhammad Ali Babar. 2011. A Systematic Review of Evaluation of Variability Management Approaches in Software Product Lines. *Inf. Softw. Technol.* 53, 4 (2011), 344–362.

[7] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord, and Judith Stafford. 2010. *Documenting Software Architectures: Views and Beyond* (2 ed.). Addison-Wesley Professional.

[8] Stéphane Ducasse and Damien Pollet. 2009. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering* 35, 4 (2009), 573–591.

[9] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *2014 IEEE International Conference on Software Maintenance and Evolution.* 391–400.

[10] Matthias Galster and Paris Avgeriou. 2011. Handling Variability in Software Architecture: Problems and Implications. In *Proceedings of the 2011 Ninth Working IEEE/IFIP Conference on Software Architecture.* IEEE Computer Society, 171–180.

[11] Matthias Galster and Paris Avgeriou. 2011. The Notion of Variability in Software Architecture: Results from a Preliminary Exploratory Study. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems.* ACM, 59–67.

[12] Matthias Galster, Danny Weyns, Paris Avgeriou, and Martin Becker. 2013. Variability in software architecture: views and beyond. *SIGSOFT Softw. Eng. Notes* 37, 6 (2013), 1–9.

[13] Joshua Garcia, Igor Ivkovic, and Nenad Medvidovic. 2013. A comparative analysis of software architecture recovery techniques. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013.* IEEE, 486–496.

[14] Hassan Gomaa. 2004. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

[15] Rich Hilliard. 2010. On Representing Variation. In *1st International Workshop on Variability in Software Product Line Architectures.* ACM.

[16] ISO/IEC/IEEE. 2011. Systems and software engineering – Architecture description. *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)* (2011), 1 –46. https://doi.org/10.1109/IEEESTD.2011.6129467

[17] Mehdi Jazayeri, Alexander Ran, and Frank van der Linden. 2000. *Software Architecture for Product Families: Principles and Practice.* Addison-Wesley.

[18] Philippe Kruchten. 1995. The 4+1 View Model of Architecture. *IEEE Software* 12, 6 (1995), 42–50.

[19] Crescencio Lima, Wesley Klewerton Guez Assunção, Jabier Martinez, William Mendonça, Ivan do Carmo Machado, and Christina von Flach G. Chavez. 2019. Product line architecture recovery with outlier filtering in software families: the Apo-Games case study. *J. Braz. Comp. Soc.* 25, 1 (2019), 7:1–7:17. https://doi.org/10.1186/s13173-019-0088-4

[20] Crescencio Lima, Wesley K. G. Assunção, Jabier Martinez, Ivan do Carmo Machado, Christina von Flach G. Chavez, and Willian D. F. Mendonça. 2018. Towards an Automated Product Line Architecture Recovery: The Apo-Games Case Study. In *VII Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS '18).* ACM, 33–42.

[21] Crescencio Lima and Christina Chavez. 2016. A Systematic Review on Metamodels to Support Product Line Architecture Design. In *Proceedings of the 30th Brazilian Symposium on Software Engineering.* ACM, 13–22.

[22] Crescencio Lima, Ivan do Carmo Machado, Eduardo Santana de Almeida, and Christina von Flach Garcia Chavez. 2018. Recovering the Product Line Architecture of the Apo-Games. In *Proceedings of the 22nd International Systems and Software Product Line Conference (SPLC '18).* ACM.

[23] Crescencio Rodrigues Lima-Neto, Mateus Cardoso, Christina Chavez, and Eduardo Almeida. 2015. Initial Evidence for Understanding the Relationship between Product Line Architecture and Software Architecture Recovery. In *2015 IX Brazilian Symposium on Components, Architectures and Reuse Software (SBCARS).* 40–49.

[24] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2016. Variability extraction and modeling for product variants. *Software & Systems Modeling* (2016), 1–21.

[25] Jens Meinicke, Thomas Thüm, Reimar Schröter, Sebastian Krieter, Fabian Benduhn, Gunter Saake, and Thomas Leich. 2016. FeatureIDE: Taming the Preprocessor Wilderness. In *Proceedings of the 38th International Conference on Software Engineering Companion.* ACM, 629–632.

[26] M. Moon, H.S. Chae, and K. Yeom. 2006. A Metamodel Approach to Architecture Variability in a Product Line. In *Reuse of Off-the-Shelf Components. ICSR 2006. Lecture Notes in Computer Science*, Morisio M. (Ed.), Vol. 4039. Springer, Berlin, Heidelberg.

[27] Edson Alves Oliveira-Junior, Itana Gimenes, and Jose Maldonado. 2008. A metric suite to support software product line architecture evaluation. In *XXXIV Conferencia Latinoamericana de Informatica.* 489–498.

[28] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques.* Springer-Verlag New York, Inc. 467 pages.

[29] Julia Rubin and Marsha Chechik. 2012. Locating distinguishing features using diff sets. In *27th IEEE/ACM International Conference on Automated Software Engineering.* ACM, 242–245.

[30] Anas Shatnawi, Abdelhak Seriai, and Houari Sahraoui. 2015. Recovering Architectural Variability of a Family of Product Variants. In *Software Reuse for Dynamic Systems in the Cloud and Beyond.* Lecture Notes in Computer Science, Vol. 8919. Springer International Publishing, 17–33.

[31] Anas Shatnawi, Abdelhak-Djamel Seriai, and Houari Sahraoui. 2016. Recovering software product line architecture of a family of object-oriented product variants. *Journal of Systems and Software* (2016).

[32] Ioanna Stavropoulou, Marios Grigoriou, and Kostas Kontogiannis. 2017. Case study on which relations to use for clustering-based software architecture recovery. *Empirical Software Engineering* 22, 4 (Aug 2017), 1717–1762.

[33] Steffen Thiel and Andreas Hein. 2002. Modeling and Using Product Line Variability in Automotive Systems. *IEEE Software* 19, 4 (2002), 66–72.

[34] Steffen Thiel and Andreas Hein. 2002. Systematic Integration of Variability into Product Line Architecture Design. In *Software Product Lines, Second Int. Conf. , SPLC 2, San Diego, CA, USA, August 19-22, 2002, Proc. (Lecture Notes in Computer Science)*, Vol. 2379. Springer, 130–153.

[35] Scott Tilley, Santanu Paul, and Dennis Smith. 1996. Towards a framework for program understanding. In *WPC '96. 4th Workshop on Program Comprehension.* 19–28.

[36] Martin Verlage and Thomas Kiesgen. 2005. Five Years of Product Line Engineering in a Small Company. In *Proceedings of the 27th International Conference on Software Engineering.* ACM, 534–543.

[37] Tao Zhang, Lei Deng, Jian Wu, Qiaoming Zhou, and Chunyan Ma. 2008. Some metrics for accessing quality of product line architecture. In *International Conference on Computer Science and Software Engineering*, Vol. 2. IEEE, 500–503.