Information and Software Technology 117 (2019) 106198

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Automatic extraction of product line architecture and feature models from UML class diagram variants

Wesley K.G. Assunção[a,*], Silvia R. Vergilio[b], Roberto E. Lopez-Herrejon[c]

[a] COINF, Technological Federal University of Paraná (UTFPR), CEP: 85.902-490, Toledo, Brazil
[b] DINF, Federal University of Paraná (UFPR), CP: 19081, CEP: 81.531-980, Curitiba, Brazil
[c] LOGTI, École de Technologie Supérieure (ÉTS), University of Quebec, H3C 1K3, Montreal, Canada

ARTICLE INFO

Keywords:
Model merging
Feature model
SPL architecture
Search-based techniques

ABSTRACT

Context: Software Product Lines (SPLs) are families of related products developed for specific domains. SPLs commonly emerge from existing variants when their individual maintenance and/or evolution become complex. Even though there exists a vast research literature on SPL extraction, the majority of the approaches have only focused on source code, are partially automated, or do not reflect domain constraints. Such limitations can make more difficult the extraction, management, documentation and generation of some important SPL artifacts such as the product line architecture, a fact that can impact negatively the evolution and maintenance of SPLs.

Objective: To tackle these limitations, this work presents ModelVars2SPL (*Model Variants to SPL Core Assets*), an automated approach to aid the development of SPLs from existing system variants.

Method: The input for ModelVars2SPL is a set of *Unified Modeling Language* (UML) class diagrams and the list of features they implement. The approach extracts two main assets: (i) Feature Model (FM), which represents the combinations of features, and (ii) a Product Line Architecture (PLA), which represents a global structure of the variants. ModelVars2SPL is composed of four automated steps. We conducted a thorough evaluation of ModelVars2SPL to analyze the artefacts it generates and its performance.

Results: The results show that the FMs well-represent the features organization, providing useful information to define and manage commonalities and variabilities. The PLAs show a global structure of current variants, facilitating the understanding of existing implementations of all variants.

Conclusions: An advantage of ModelVars2SPL is to exploit the use of UML design models, that is, it is independent of the programming language, and supports the re-engineering process in the design level, allowing practitioners to have a broader view of the SPL.

1. Introduction

One of the key factors for improving quality, productivity and consequently reducing costs in software development is software reuse - the process of creating software systems from existing artifacts rather than building them from scratch [1]. Any artifact built during the software development can be reused, including source code, design models, test cases, etc.

A common industrial practice is the *opportunistic reuse*, also known as clone-and-own reuse, copy-and-paste reuse, or ad hoc reuse [2]. When there is demand for a new system functionality, existing software artifacts are cloned/copied and adapted/modified to fulfill the new requirements. Opportunistic reuse arises from a variety of organizational and technical reasons. Among them, because it is an easy way to reuse software, does not require an upfront investment, and quickly obtains good short-term results. However, opportunistic reuse results in extensive refactoring and adds technical debt, leading to unanticipated behavior, violated constraints, conflicts in assumptions, fragile structure, and software bloat [3]. The simultaneous maintenance and evolution of a typically large number of individual system variants is a complex activity [4]. Practitioners commonly do not have artifacts at a high level abstraction that enable their understanding of how the different implementations are spread along the product variants. In this scenario a systematic reuse approach is necessary. A proven way to tackle this problem is adopting a Software Product Line approach [5].

A *Software Product Line (SPL)* is a set of software products that share common parts, and is designed for a specific domain [6,7]. SPL approaches allow reuse of a common infrastructure composed of *core as-*

* Corresponding author.
E-mail addresses: wesleyk@inf.ufpr.br (W.K.G. Assunção), silvia@inf.ufpr.br (S.R. Vergilio), roberto.lopez@etsmtl.ca (R.E. Lopez-Herrejon).

https://doi.org/10.1016/j.infsof.2019.106198
Received 4 March 2019; Received in revised form 30 September 2019; Accepted 4 October 2019
Available online 8 October 2019

sets, which are shared to create different product variants. Two kinds of assets are identified in Software Product Line Engineering, the *common assets* (also known as commonalities) reused in all products, and the *variable assets* (also known as variabilities) related to those features that are provided only by some products. Examples of core assets are components, domain models, design models, requirements statements, documents, plans, test cases or any other useful reusable artifact of the software production process. One of the main core assets of an SPL is its architecture. The *Product Line Architecture (PLA)* helps the generation of the products and allows practitioners to better understand, maintain, and evolve a system family. Other fundamental core asset is the variability model, such as the *Feature Model (FM)*, which has become the *de facto* standard artifact used for *variability modeling* and *variability management* [5]. FMs express common and variable characteristics of a product family and are used to support stakeholders in making technical and managerial decisions about the portfolio of products. To generate and maintain these artifacts is fundamental to ease the evolution and maintenance of the SPL.

In many organizations it is common the existence of a large base of artifacts already developed. In this case, the SPL core assets are more easily obtained by using an extractive approach, pointed out by some authors as the most common way to systematize the software reuse with SPLs [8,9]. This approach encompasses the re-engineering of existing system variants, leading to a systematic reuse and easier maintenance and evolution, because the systems are not maintained or evolved individually but as a group considering both commonalities and variabilities. However, the re-engineering process is a complex activity which involves many steps and deals with a great diversity of artifacts in different levels of abstraction. This process requires extensive human effort and is an error-prone task [10].

The extraction of SPLs from system variants has attracted interest from industry and academia. In the research literature, we can find many works on this topic [8,9]. Such works were analyzed and classified in mapping studies [9,11]. Thanks to these mappings we identified some limitations of the existing works, which are mainly related to: the type of used artifacts, the phases of the re-engineering process addressed, the type of feature management, and the automated tool support.

We found that most pieces of work have focus exclusively on source code [12]. Hence, there is a lack of approaches considering other types of artifacts, such as design models. Design models are essential artifacts to understand, develop, evolve and maintain SPLs. We also observed that existing studies address specific phases of the re-engineering process, most of them have focus only on the traceability between features and implementation artifacts [13–17]. There is a lack of approaches and tools to widely support the re-engineering process. Most approaches require extensive human expertise and participation [18–21] and generally do not consider domain constraints among features [22,23]. An adequate feature management is important to deal with domain constraints, but this can be challenging due to the large number of possible feature combinations and the lack of explicit information about features interactions/dependencies.

To overcome these limitations, our work addresses the problem of automatically extracting SPL core design assets from model variants, covering the re-engineering phases (detection, analysis, and transformation) and taking into account domain constraints existing in the variants under consideration.

The solution for this problem is neither simple nor trivial. An automated approach for the re-engineering process requires some fundamental phases such as detection of features, organization of variabilities, and transformation of artifacts; that have different goals and deal with specific information and artifacts. In some cases, the input artifacts have implicit information or are partially incomplete, for example, the existing domain constraints in terms of feature combinations are not clearly described, or only a small set of variants is available. These situations require a well designed re-engineering strategy. Furthermore, design models are commonly complex artifacts used to describe system structures. Consequently, dealing with many of these artifacts, each one representing a different variant, can be challenging.

Complex problems like this have been effectively and efficiently solved in the Search-Based Software Engineering (SBSE) field [24], where software engineering problems are modeled as optimization problems and then solved with search-based techniques. Search-based techniques are beneficial for the automation of several tasks of the re-engineering process. For such tasks, developers most times can recognize a good solution but to obtain it can be very difficult.

In this paper we propose an approach, called ModelVars2SPL (*Model Variants to SPL core assets*), that given as input the class diagrams and feature sets of the variants, allows automatic extraction of two main SPL core design assets - FMs and PLAs.

ModelVars2SPL has the following characteristics:

(i) It uses UML models as both input and output, differently of some related approaches, which are code-based. The input for the approach is a set of UML class diagram variants, artifacts that are commonly available in the software development organizations. UML is the most used language in industry to represent software architectures [25,26]. The output is the FM and a UML-based PLA, composed of a class diagram and feature annotations. Such output eases communication, comprehension, planning, maintenance, and future evolution of the SPL;
(ii) It covers three common phases of the re-engineering process, namely detection, analysis, and transformation, by generating outputs for each phase, allowing practitioners to follow and understand the process;
(iii) Each step of the approach is automated and the outputs are produced with minimal human intervention. The most complex tasks are solved with search-based techniques. The use of such techniques is justified because such tasks are associated to a large search space, that is, there exist a great number of solutions representing different combinations of features and relationships beweem them. In many situations, information are missing or are poorly described;
(iv) It was implemented with widely used tools to fit better in industrial scenarios.

The main contribution of ModelVars2SPL is to aid the process of re-engineering system variants into an SPL by automatically generation of FMs and PLAs, two main product line core design assets. Differently from works existing in the literature, ModelVars2SPL is model-based, which makes it independent of programming languages and permits focus on the SPL design. This can make easy architects, engineers, and developers to communicate, plan, reason, and estimate on how to extract an SPL from existing system variants. The defined approach encompasses an automatic process that reduces the low-level manual activities and enables software engineers to keep their focus on high-level activities, such as managerial decisions, time-to-market, and customer satisfaction.

To evaluate ModelVars2SPL, we conducted an evaluation with ten applications composed of different number of variants. The results show that ModelVars2SPL can automatically produce SPL design artifacts that well-represent the variants input, which aid practitioners to make decisions and plan the extraction of SPLs from existing systems. The FMs generated have optimal values of precision, recall and variability safety when compared to the input variants. The PLAs obtained have proper similarity to the existing class diagram variants. In addition, the number of candidate solutions and the required runtime show the approach is suitable to application in practice.

The remainder of this paper is as follows. Section 2 reviews related work. Section 3 introduces ModelVars2SPL. Section 4 describes the evaluation of ModelVars2SPL. The results are presented in Section 5 and discussed in Section 6. Section 7 discusses usefulness of the artifacts generated by ModelVars2SPL. Section 8 contains the final remarks and future work.

2. Related work

In this section we present related work and highlight their limitations and gaps, which we focus on filling with ModelVars2SPL. The starting point to identify relevant studies was two mapping studies [9,11]. We also considered a catalog of Extractive Software Product Line Adoption case studies (ESPLA) [12]. This catalog was constructed from studies found in literature and is maintained by the SPL community.[1]

When dealing with UML models, most of studies are focused on feature identification, which is only the first step of the re-engineering process towards an SPL. For example, Martinez et al. propose an approach called MoVaC, which identifies commonalities and variabilities among model variants [13]. As results, MoVaC presents a visualization of existing features among the variants and the model elements that belong to each feature. Another work designed to identify variability among features, focusing on models, is proposed by Wille et al [14]. However, this work deals specifically with state-charts, due to the domain they are interested in. Rubin et al. also have focus on comparing UML state-charts with the goal of refactoring them into an SPL [15]. These authors perform identification of similar model elements to generate a single model representation, without describing features specific implementations or feature interactions, i.e. they do not highlight commonality and variability between model variants.

In the work of Ziadi et al., features of product variants are identified supported by class diagrams. However, the input artifact for their approach is the source code. The class diagrams are used only as a simplified representation of the implementation artifacts [16]. A method that uses UML models as source of information to extract FMs is presented by Mefteh et al. [17]. Here, the authors extract an initial version of an FM from UML use case diagrams and natural language description of existing scenarios. Our approach differs from this work because we consider UML class diagrams and feature sets (i.e. configurations represented as sets of selected and unselected features).

Only two studies deal with the entire re-engineering process of UML models variants to an SPL [27,28], but these studies deal with illustrative small size subject systems, namely Microwave Oven and Washing Machine. However, these studies only have focus on UML models. The organization of existing features in an FM is not considered. Furthermore, in the ESPLA catalog only 13 case studies deal with models of which eight are UML models.

Thus, in this regard, we identified two important gaps: *(i) few studies deal with artifacts other than source code (130 out of 60), and a small fraction among them uses UML models (only 13), (ii) UML models research focused mostly on the detection step of the re-engineering process.* To address these gaps, we designed ModelVars2SPL to extract SPL core design assets from UML model variants and to cover all the re-engineering phases (detection, analysis, and transformation).

With the goal of identifying core reusable requirements Reinhartz-Berger and Kemelman proposed an automated extractive method named CoreReq [22]. From product requirements written in natural language, CoreReq clusters similar requirements, analyses variability among them, and generates core requirements by using natural language processing techniques. However, to the best of our knowledge, they do not take into account the domain constraints. In an SPL, these requirements will be implemented as features, which are mostly represented in FMs. The reverse engineering of FMs is subject of the work of She et al. [23]. Using as input a set of feature names, feature descriptions, and features dependencies, some heuristics are applied and an FM is constructed. However, their work does not focus on dealing with system variants, where each variant implements a different set of features. In addition, despite using features dependencies, they do not infer other possible constraints that domain requirements could impose. Hence, this constitutes a third gap we identified: *(iii) discovery of domain constraints existing among features and representing them into an FM.* Our work ModelVars2SPL addresses this a gap by identifying features constraints/dependencies using a search-based technique to reverse engineering them.

Bayer et al. introduce an approach, called PuLSETM-DSSA, to integrate software artifacts in different abstraction levels to obtain a reference architecture [18]. The reference architecture gives guidance for migrating the system variants into an SPL. Differently from our approach, PuLSETM-DSSA is not fully automated, requiring human expertise.

The work of Kumaki et al. has a goal similar to our work, which is to generate a variability model and an architecture [19]. The input of their approach is also a set of UML class diagram variants, but instead of using feature sets, they use a set of requirements. Their approach uses overlap analysis to identify common and variable sentences in the requirements. The same overlap analysis is done in the class diagram variants. An algorithm based on vector space model is applied to recommend traceability between requirement sentences and class diagram elements. Using this traceability, an SPL expert develops the variability model and a reference architecture that best represents the variants. In contrast with our work, theirs requires human intervention.

Martinez et al. propose an approach called MoVa2PL to support model-based adoption of SPLs [20]. MoVa2PL first identifies common and variable blocks of model elements from a set of model variants, then these blocks of model elements are mapped manually to their corresponding features. The model-based SPL is obtained by describing variabilities using the Common Variability Language [21]. Taking into account the re-engineering process steps, MoVa2PL deals only with the feature identification activity. Hence, the analysis and transformation steps are not the focus of their work.

Taking into account the aforementioned, the fourth gap identified is: *(iv) extensive manual effort for conduction the entire re-engineering process.* In ModelVars2SPL, the re-engineering steps are automated and the outputs are produced with minimal human intervention.

Based on the gaps found in the literature, we designed the ModelVars2SPL approach, which deals with UML models and feature sets covering the entire re-engineering process. In the step of analysis, ModelVars2SPL discovers domain constraints among features and represents them in the FM. Furthermore, our approach presents an automated solution for each step of the re-engineering process, reducing the need of manual effort to generate the FM and PLAs. Next we present the details of our approach.

3. ModelVars2SPL

ModelVars2SPL is an approach to extract SPL core design assets from existing model variants. The main goal is to aid the re-engineering process, offering automatic support to its three main phases [9]: (i) *Detection phase*, devoted to detect the variabilities and commonalities among existing [9] products; (ii) *Analysis phase*, involves the organization of discovered variabilities and commonalities; and (iii) *Transformation phase*, uses the existing artifacts and the variability model to create the SPL architecture, which is the basis for managing implementation artifacts. We first present an overview of our approach followed by detailed explanation and illustration of each of its steps.

3.1. Approach overview

ModelVars2SPL includes four steps, represented as activities, as illustrated in Fig. 1. Each step produces different outputs that are used by the other steps or that are intended to support practitioners in all the re-engineering activities.

The input of ModelVars2SPL is a set of variants. Each variant consists of two parts: (i) a *UML class diagram* that provides a static view of the structure of the system variant, and (ii) a *feature set* that denotes the configuration of features provided by the variant. To illustrate the

[1] On the publication of this paper ESPLA catalog (https://but4reuse.github.io/espla_catalog/). Last updated in August 2018.

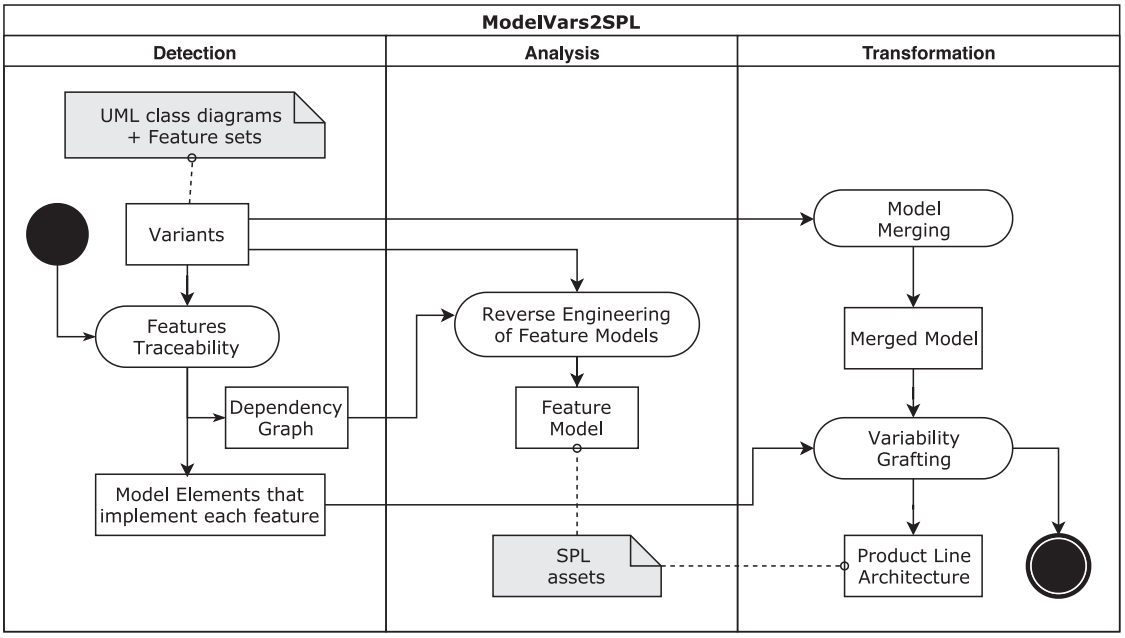


Fig. 1. Input, steps, and output overview of ModelVars2SPL.

input and explain the steps of ModelVars2SPL, Fig. 2 presents an illustrative example of a Banking System (BS), and shows four UML class diagram variants[2] and the corresponding feature sets, which are in the captions. BS is a running example encompassing UML model variants for different scenarios [16]. In this example we considered four models created by opportunistic reuse. The first variant BS01 (Fig. 2(a)) has only basic and common features (`Base`) of a banking system. The variant BS05 (Fig. 2(b)) is an extension of BS01, which was modified to deal with information related to currency conversion (`Converter`) and `Consortium`. BS06 (Fig. 2(c)) includes a new requested feature that is the support of a limit in the withdrawal (`WithdrawLimit`), also has the `Converter`, but does not need `Consortium`. The last variant, BS07 (Fig. 2(d)) was based on the previous model, where the feature `Converter` was removed and the feature `Consortium` was included.

In a context where practitioners do not have the UML class diagrams available, tools such as Eclipse MoDisco[3] can be used to reverse engineer the diagrams. There is no required characteristc or restriction for modeling/reverse engineering the model variants, they just need to represent the structural perspective of system variants. In the scope of this work, we do not take into account behavioral or dynamic aspects of the existing system.

The steps of ModelVars2SPL are shown in the activity diagram of Fig. 1. The phases of the re-engineering process associated to each step are represented by swim lanes. The phases of detection and analysis encompass only one step each, and the transformation phase two. A brief description of each step is presented next.

Step 1. *Features traceability* aims at identifying the model elements that implement each feature. The input for this step are both the UML class diagram and the feature set of each variant. The traceability is identified by analyzing overlaps between model elements and overlaps between feature sets of different variants. This step produces two outputs: (i) traceability links between model elements and the features they implement, and (ii) a dependency graph that represents the relationship between features.

Step 2. *Reverse engineering of feature models* applies a multi-objective search-based technique to obtain an FM that best represents the feature sets. Reverse engineering of FMs is a complex problem, since there is a huge number of possible feature combinations, and existing constraints between features are not explicitly described, facts that justify the use of search-based techniques. In this step, only the feature sets of the input variants are used. Additionally, the dependency graph produced in the previous step is also used as input, with focus on generating FMs that preserves constraints of dependencies between model elements.

Step 3. *Model merging* is responsible for combining multiple class diagram variants from the input. The goal is to synthesize an UML class diagram that contains all possible model elements present in different variants. The merging of multiple models is a NP-hard problem, once the comparison among all model elements for a big set of variants leads to a huge search space [29]. This task is performed by a search-based technique that evaluates the merged class diagrams considering the number of differences from the candidate individual to all input model variants. At the end of the search process a merged model is obtained.

Step 4. *Variability grafting* aims at enriching the merged class diagram obtained in the previous step with information about features and variability to produce a PLA. Each model element of the merged UML class diagram is annotated with the feature, or set of features, it corresponds. This task is done by adding a UML comment to each model element.

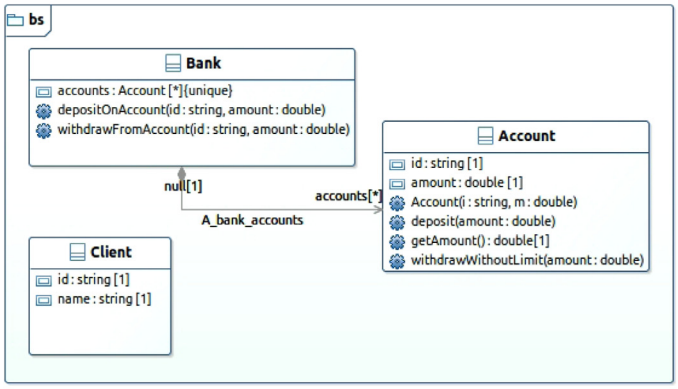
Based on the input presented in Figs. 2, 3 presents an example of SPL design artifacts generated, outputs of ModelVars2SPL. The FM on the left side of the figure shows the possible combination of features. On the right of the figure we present the PLA, composed of the UML class diagram, that has model elements from all input variants (i.e., a global model), and the feature annotations. These feature annotations are illustrated in the middle of the figure. They allow observing the traceability between features and model elements, as depicted with arrows in the figure. For example, attribute `accounts: Account[*]{unique}` of `Bank` class is part of the implementation of feature `Base`, the class `Consortium` implements the feature `Consortium`, and so on.

3.2. Features traceability

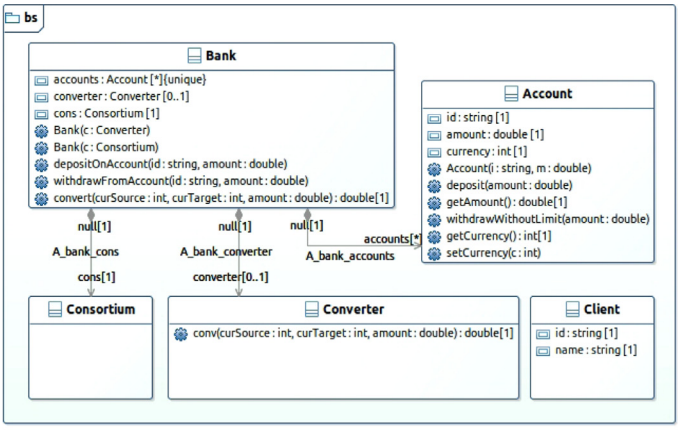
This step includes two activities: the decomposition of the model variants in atomic elements and traceability discovery. Atomic elements

[2] The graphical representations of UML models were obtained using UML Designer: http://www.umldesigner.org/.

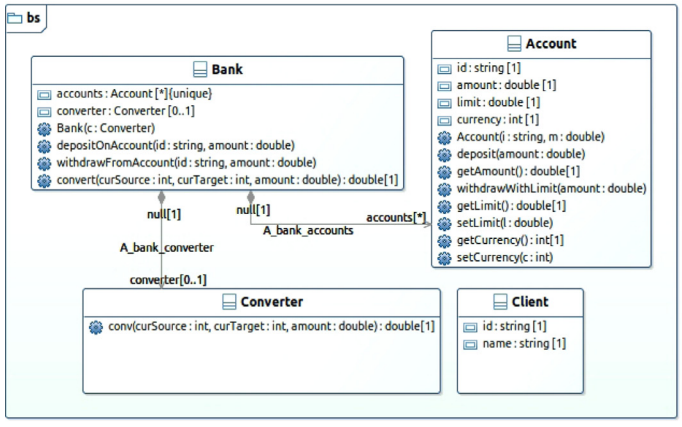
[3] https://eclipse.org/MoDisco.



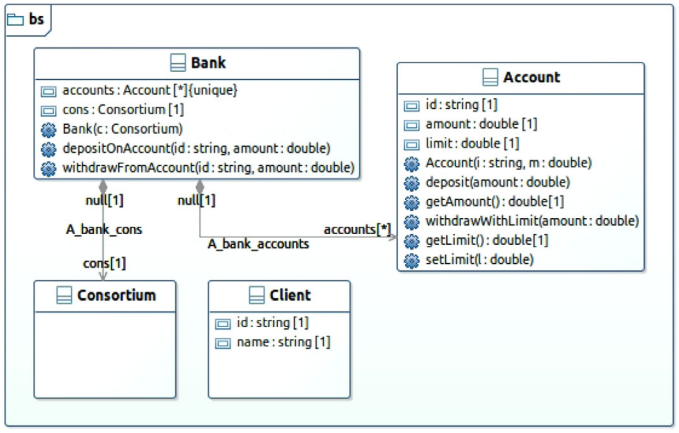
(a) BS01: {Base}



(b) BS05: {Base, Consortium, Converter}



(c) BS06: {Base, WithdrawLimit, Converter}



(d) BS07: {Base, WithdrawLimit, Consortium}

Fig. 2. Four variants of Banking System used as input to illustrate the steps of ModelVars2SPL.

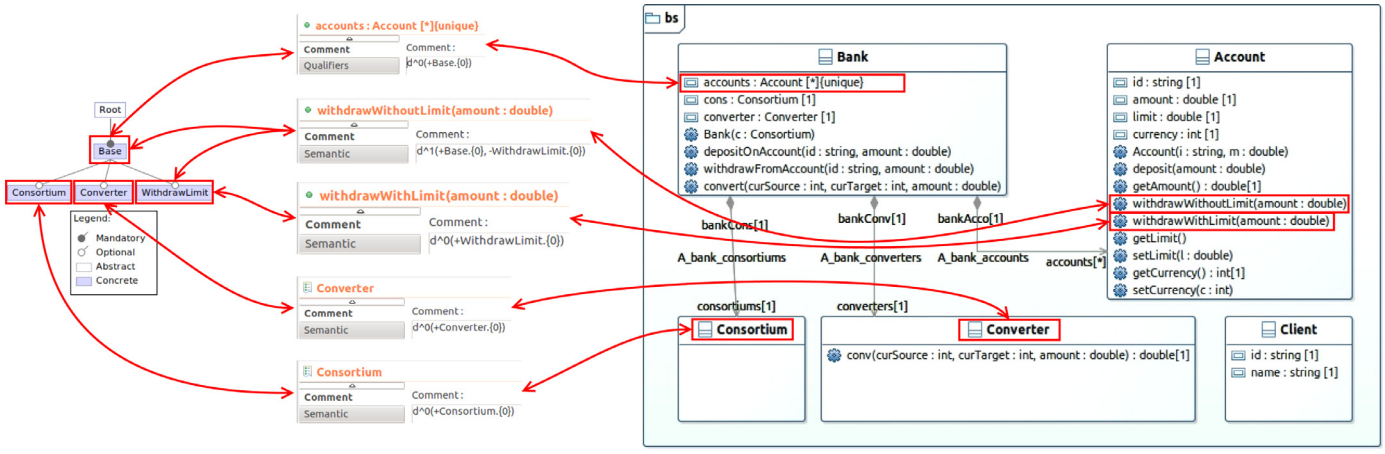


Fig. 3. Example of FM and PLA Banking System obtained by ModelVars2SPL. Arrows illustrated the feature annotations.

are parts of the models with low granularity that are relevant to the re-engineering process. We consider as atomic model elements: classes, interfaces, attributes, methods/operations, and all class diagrams relationships.

Regarding the traceability discovery, an ideal result is to identify each set of elements that implement each distinct feature. However, in real cases, some implementation elements only appear when two or more features interact. These are elements responsible for providing a proper feature interaction. Furthermore, some implementation elements may appear in cases where a feature is absent.

Based on the possibility of features interactions and absent features, our approach relies on the terminology presented by Linsbauer et al. [30] to distinguish two kinds of *modules* in the system variants:

Definition 1 (Base Module). A *base* module $m = d^0(+c.\{v\})$ implements a feature regardless of the presence or absence of any other features. This module is represented by a derivative of order 0 (d^0), where $+c$ is

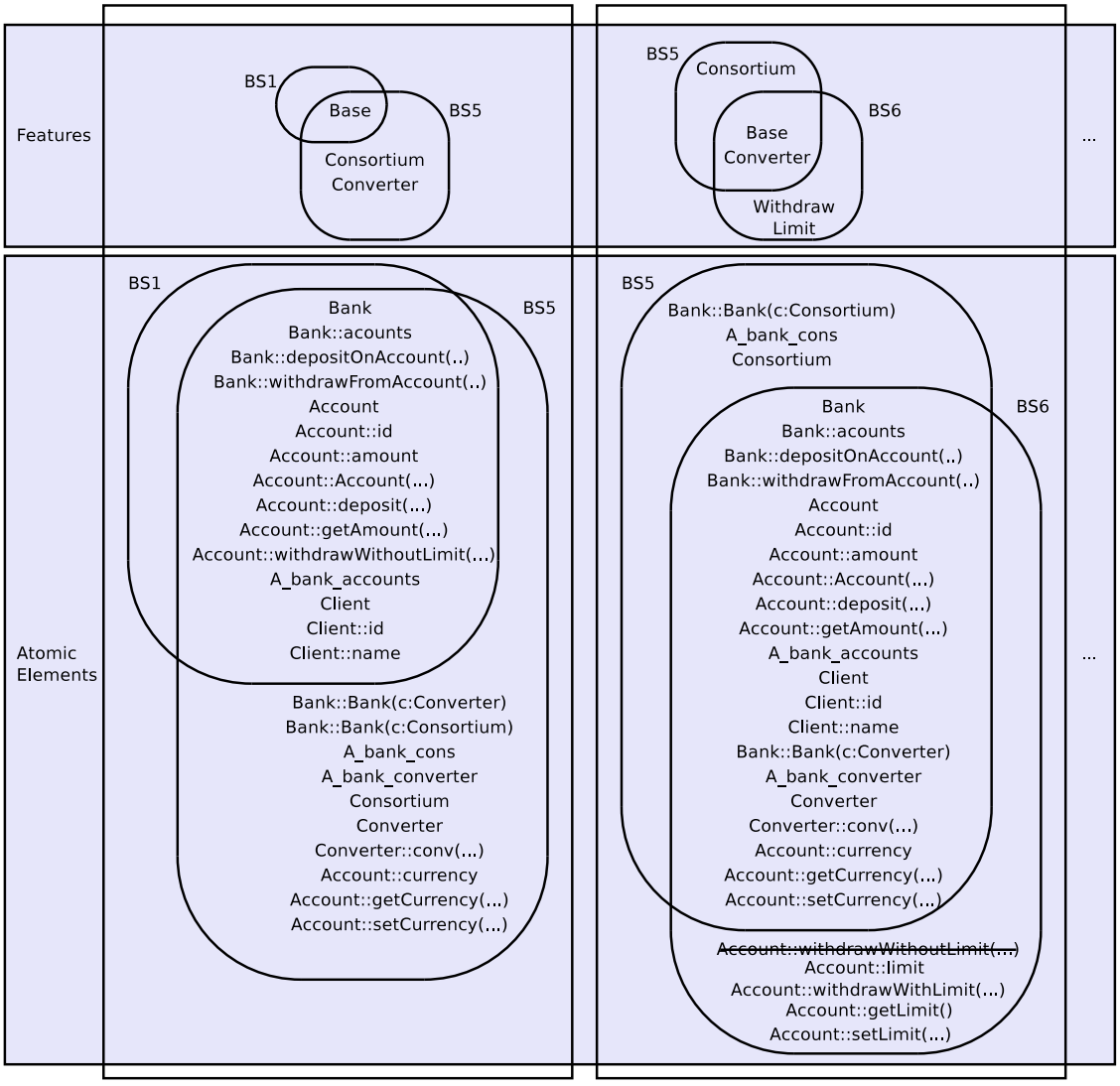


Fig. 4. Example of overlap analysis for BS1 vs BS5 and BS5 vs BS6.

a feature, and $\{v\}$ is an optional number related to the version of the variant when the feature was included.

Definition 2 (Derivative Module)**.** A *derivative* module $m = d^n(c_0.\{v\}, c_1.\{v\}, \ldots, c_n.\{v\})$ implements feature interactions, where c_i is +F (if feature F is selected) or - F (if not selected), and n is the order of the derivative.

The idea is to automatically produce a set of traces between both types of modules and artifacts that implement them (i.e., atomic elements). The traces are produced by matching atomic elements overlaps and feature overlaps. Each variant is incrementally analyzed comparing overlaps already analyzed with new variant artifacts provided.

To illustrate the traceability discovery, let us consider overlap analysis among three models of our running example. Fig. 4 presents the feature sets and the atomic model elements of BS1, BS5, and BS6. Firstly BS1 and BS5 are analyzed, as presented on the left side of the figure. These variants have in common the features `Base` and 15 model elements. Based on this analysis we can assume that the feature `Base` is implemented by these 15 model elements. Furthermore, only BS5 has `Consortium` and `Converter`, and 10 model elements. At this point, we cannot separate the model elements of each these features. On the analysis of BS5 and BS6, on the right side of the figure, the variants have in common the features `Base` and `Converter`, and 21 model elements. Considering this, and the previous analysis, we can refine the traceability and assume which are the models elements which implement `Converter`, since we had the model elements that compose `Base`. We can corroborate this analysis with the analysis of `Consortium`, which is in BS5, but not in BS6. Another important analysis is regarding the feature `WithdrawLimit`. We can observe that `WithdrawLimit` in BS6 implies the exclusion of the method `Account::widthdrawWithoutLimit(...)` in the variant. Then, we can assume that the inclusion of the feature `WithdrawLimit` leads to the exclusion of one method, and inclusion of three methods and one attribute. This form of analysis is performed among all variants to compute the traceability of all features and their interactions.

Another relevant information for our approach is the relationships (e.g., dependencies) existing between elements that implement the features. We obtain this information during the traceability discovery and represent it using a dependency graph. The following definitions are based on our previous work [31].

Definition 3 (Dependency)**.** A dependency establishes a requirement relationship between two sets of modules and it is denoted with a three-tuple `(from, to, weight)`, where `from` and `to` are a set of modules (or module expressions) of the related modules, and `weight` ex-

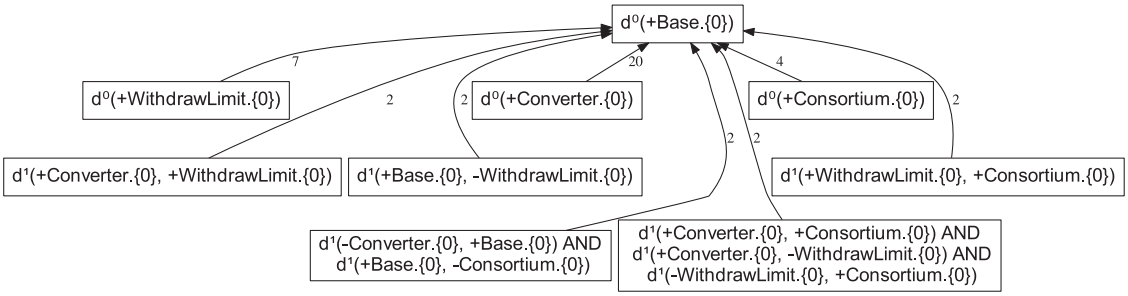


Fig. 5. Example of dependency graph.

presses the strength of the dependency, i.e., the number of dependencies of structural elements in modules `from` on structural elements in modules `to`.

Definition 4 (Dependency Graph)**.** A dependency graph is defined as a set of dependencies, where each node in the graph corresponds to a set of modules (or module expressions), and each edge corresponds to a dependency as defined above. Edges are annotated with natural numbers that represent the dependencies weights.

Fig. 5 shows the dependency graph of our illustrative example considering its four variants. In this example we observe four base modules and five derivative modules.

3.3. Reverse engineering of feature models

The step of Reverse Engineering of FMs of our approach is based on our research that relies on a search-based technique [31]. This task requires two inputs: (i) a set of feature sets from the input variants, and (ii) the dependency graph obtained in the previous step.

Each solution in the search-based technique is represented by a meta-model with the nodes *Root, Mandatory, Optional*, and *GroupedFeature* that are children of *Feature*. A tree can have only one *Root*, and the features *Mandatory* and *Optional* have the cardinality of zero or more. The number of *Alternative* and *Or* groups is arbitrary; however, each group must have at least one *GroupedFeature*. An FM has exactly one *ConstraintSet* that describes the propositional formula in Conjunctive Normal Form (CNF). A *Constraint* has one *OrClause* that has at least one *Literal*. A literal can be an *Atom* or a *Not*.

Our multi-objective search-based technique uses three metrics to evaluate solutions. To describe these metrics, let us consider $\mathcal{FM}$ as the universe of candidate feature models, $\mathcal{SFS}$ the universe of feature sets, and *sfs* the feature sets from the system variants, we define the auxiliary function called *featureSets*:

Definition 5 (featureSets)**.** Function *featureSets* returns the set of feature sets denoted by a feature model.

$$featureSets : \mathcal{FM} \rightarrow \mathcal{SFS}$$

Taking into account the existence of dependencies between different features, the function *holds* checks if a dependency is not violated in a feature set. As follows:

Definition 6 (holds)**.** Function *holds*(*dep, fs*) returns 1 if dependency *dep* holds on the feature set *fs* and 0 otherwise. A dependency *dep* holds for a feature set *fs* if:

$$\left(\bigwedge_{f \in fs.sel} f \wedge \bigwedge_{g \in \overline{fs.sel}} \neg g \right) \Rightarrow (dep.from \Rightarrow dep.to)$$

Considering the two auxiliary functions presented above, we introduce the three metrics used as objective functions of our search-based technique. The first two functions are based on feature sets, and the third function is based on dependencies between features.

Definition 7 (Precision (P))**.** Precision expresses how many of the feature sets denoted by a reverse engineered feature model *fm* are among the desired feature sets *sfs*.

$$precision(sfs, fm) = \frac{|sfs \cap featureSets(fm)|}{|featureSets(fm)|}$$

Definition 8 (Recall (R))**.** Recall expresses how many of the desired feature sets *sfs* are denoted by the reverse engineered feature model *fm*.

$$recall(sfs, fm) = \frac{|sfs \cap featureSets(fm)|}{|sfs|}$$

Definition 9 (Variability Safety (VS))**.** Variability Safety expresses the degree of *variability-safety* of a reverse engineered feature model *fm* with respect to a dependency graph *dg*.

$$variabilitySafety(fm, dg) = \sum_{dep \in dg} dep.weight \times \left(\frac{\sum_{fs \in featureSets(fm)} holds(dep, fs)}{|featureSets(fm)|} \right)$$

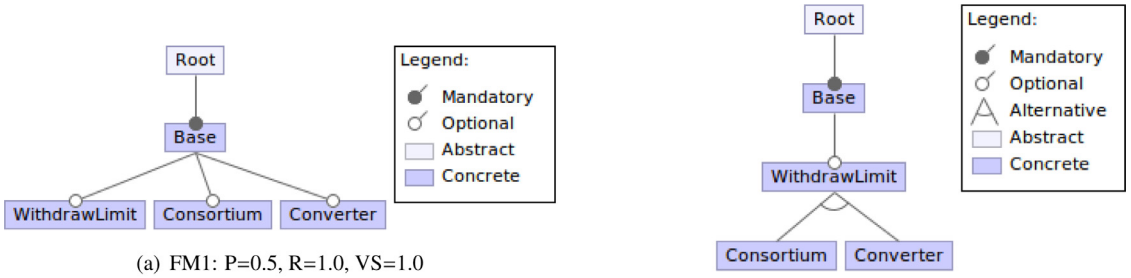
The three fitness functions presented above are designed to keep the computed values between 0 and 1. Moreover, the goal is to maximize the values for all functions. An ideal solution is P = 1.0, R = 1.0, and VS = 1.0.

The operations used in this step to generate candidate solutions adhere to the following constraints to guarantee the semantics of FMs:

- Each feature is identified by its name, so every feature appears exactly once in the FM tree;
- All FMs have a fixed set of feature names, so in different FMs only the relations between features are different;
- *Cross-Tree Constraints (CTCs)* can only be either *requires* or *excludes*, i.e., exactly two literals per clause with at least one being negated;
- CTCs must not contradict each other, i.e., the corresponding CNF of the entire constraint set must be satisfiable;
- There is a maximum number of CTCs (given as a percentage of the number of features) that must not be exceeded.

The *crossover* operator generates offspring following the meta-model representation and in conformance to the domain constraints. The crossover process is:

1. Initialize the offspring with the root feature of $Parent_1$. If the root feature of $Parent_2$ is a different one then it is added to the offspring as a mandatory child feature of its root feature.
2. Traverse the first parent depth first starting at the *root* node and add to the offspring a random number *r* of features that are not already contained by appending them to their respective parent feature already contained in the offspring using the same relation type between them (the parent feature of every visited feature during the traversal is guaranteed to be contained in the offspring due to the depth first traversal order).
3. Traverse the second parent exactly the same way as the first one.



(a) FM1: P=0.5, R=1.0, VS=1.0

(b) FM2: P=1.0, R=0.75, VS=1.0

Fig. 6. Example of reverse engineered FMs.

4. Go to Step 2 until every feature is contained in the offspring.

The second child is obtained by swapping the order of the parents. The CTCs offspring are assigned randomly from a set of constraints obtained by merging all the constraints of both parents. First a subset of CTCs are selected and assigned to the first offspring and then the remaining to the second offspring.

The *mutation* operator applies minor modifications in the tree or in the CTCs of the FM. A probability value is used to choose if the change is applied in the tree part, in the CTCs, or in both. The modifications applied in an FM are:

- Mutations performed on the tree:
  - Randomly swaps two features in the feature tree;
  - Randomly changes an *Alternative* relation to an *Or* relation or vice-versa;
  - Randomly changes an *Optional* or *Mandatory* relation to any other kind of relation (*Mandatory, Optional, Alternative, Or*);
  - Randomly selects a sub-tree in the feature tree and puts it somewhere else in the tree without violating the metamodel or any of the domain constraints.
- Mutations performed on the CTCs:
  - Adds a new randomly created CTC that does not contradict the other CTCs and does not already exist;
  - Randomly removes a CTC.

The candidate FMs chosen to participate in the crossover/mutation are selected by standard tournament selection. The starting point for the search-based technique is a set of initial FMs created by generating random feature trees and random CTCs. Domain constraints are also taken into account to generate random FMs.

To illustrate the reverse engineering of FMs we recall the BS variants (Fig. 2), where the set of feature sets for our illustrative example is presented in Table 1. The dependencies of this illustrative example are presented in Fig. 5, however we represented the dependency graph with normalized values in Table 2. This normalization keeps the sum of all weights of the graph equal to 1.0, enabling a better interpretation of the values in the optimization process.

Table 1
Feature sets of the illustrative example.

Variants	Feature sets			
	Base	Consortium	Converter	WithdrawLimit
BS01	✓			
BS05	✓	✓	✓	
BS06	✓		✓	✓
BS07	✓	✓		✓

The information of Tables 1 and 2 are provided as input for the search-based technique. Fig. 6 presents two FMs reached by the algorithm.[4] Briefly, we can observe that both FMs have VS equal to 1.0, which means they do not violate any dependency of the dependency graph. Regarding the objectives of precision and recall, there is a trade-off between the solutions. FM1 (Fig. 6(a)) has R=1.0, which means the four configurations of Table 1 are possible with this FM, however, the same FM denotes more products than desired, decreasing the values of precision (P) to 0.5. On the other hand, the value of precision for FM2 (Fig. 6(b)) is 1.0, therefore all products of this FM are in the desired feature sets, however it represents only three products, missing one desired configuration, leading to a recall value equal to 0.75. The software engineer can choose the FM that best fits his/her needs. With this example we observe the benefits on having a multi-objective solution.

3.4. Model merging

This is the first step towards the PLA. The goal of this step is to generate a global class diagram with the highest similarity to the input variants. To reach this highest similarity, the global class diagram must have as many as possible model elements, which are spread among the variants. To find such diagram is an optimization problem, because a great number of possible diagrams exists [29]. Hence, we apply in this

[4] The FMs and configurations presented are created with FeatureIDE: http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/.

Table 2
Dependency matrix of the illustrative example.

From	To	Weight	Normalized
d^0(+WithdrawLimit.{0})	d^0(+Base.{0})	7	0.170731707
d^0(+Converter.{0})	d^0(+Base.{0})	20	0.487804878
d^0(+Consortium.{0})	d^0(+Base.{0})	4	0.097560976
d^1(+Converter.{0}, +WithdrawLimit.{0})	d^0(+Base.{0})	2	0.048780488
d^1(+Base.{0}, −WithdrawLimit.{0})	d^0(+Base.{0})	2	0.048780488
d^1(+WithdrawLimit.{0}, +Consortium.{0})	d^0(+Base.{0})	2	0.048780488
d^1(−Converter.{0}, +Base.{0}) AND d^1(+Base.{0}, −Consortium.{0})	d^0(+Base.{0})	2	0.048780488
d^1(+Converter.{0}, +Consortium.{0}) AND d^1(+Converter.{0}, −WithdrawLimit.{0}) AND d^1(−WithdrawLimit.{0}, +Consortium.{0})	d^0(+Base.{0})	2	0.048780488
	Total:	41	1.0000

step a search-based technique, using as basis our previous work [32]. The technique searches for a class diagram that has the most similar structure with all the model variants provided as input.

The candidate solutions in this step use the same structure of the step Features Traceability, a UML class diagram following UML2 metamodel. We selected a representation which allows the UML models being compared, modified, and saved.

The candidate solutions are evaluated according to their differences to the UML class diagram variants. Common UML model comparing tools are able to deal with only two models at once (or three models if we include an ancestor model). However, to evaluate a candidate solution we have to compute differences from one UML class diagram to many UML class diagram variants. Considering this, the proposed fitness function is composed of the sum of differences from one model to all input model variants. The definition below presents the fitness function called *Model Similarity*. The function *diff* represents the number of differences found by using common UML model comparing tools. We sum only the set of differences indicating that the elements exist in the variant *v* but are missing on the *candidate_model*.

Definition 10 (Model Similarity (MS))**.** Model Similarity expresses the degree of similarity of the candidate architecture model to a set of model variants.

$$MS = \sum_{v \in Variants} diff(candidate\_model, v)$$

The set of differences found between the models is also used to perform crossover and mutation. The selection of solutions to apply crossover and mutation is done by the tournament strategy. For the *crossover* the starting point are two candidate models. From these two models we generate two children: one with the differences merged and one without the differences. For instance, let us consider two parent models X and Y. The children will be:

- *Crossover Child Model 1*: this model has the differences between its parents merged. For example, the elements of X that are missing on Y are merged in this latter, or vice versa. This operator is commutative.
- *Crossover Child Model 2*: this child is generated by removing the differences between the parents. For example, the differences of X that are missing on Y are removed, or vice versa. This operator is commutative.

The strategy adopted to *Child Model 1* aims at creating a model that has more elements, going to the direction of the system architecture. On the other hand, the strategy used for *Child Model 2* has the goal of eliminating possible conflicting elements from a candidate architecture.

Mutation applies only one modification in each model parent. The starting point of mutation operator is two candidate architectures, and the result is also two children. Let us consider any parent models X and Y. The children are:

- *Mutation Child Model 1*: the first child is created by merging one difference of the model Y in the model X. After randomly selecting one element of the model Y, but missing on the model X, this element is added in the model X.
- *Mutation Child Model 2*: here, the same process described above is performed, however, including one element of the model X in the model Y.

The mutation process can select a difference that is owned (i.e., is part of) another difference. In such cases, the entire owning difference is moved to the child. For example, when a mutation selects a parameter owned by an operation, the entire operation is moved to the child.

The initial population of model candidates is created by copying the input UML models variants. The only constraint is that all the input models must be included in the initial population at least once. When the population size is greater than the number of input variants, multiple copies of each variant are included.

An example of output generated by our model merging approach is presented in Fig. 7, considering the four variants of the illustrative example (Fig. 2). We can observe that all model elements of the input variants are merged in one global UML class diagram.

3.5. Variability grafting

In ModelVars2SPL, a PLA is a structural representation of model variants with annotations regarding the variabilities and features of the variants. To generate such representation, the traceability discovered in the step Features Traceability is used to graft information regarding the features in the merged model obtained in the previous step. The task of variability grafting is done by indicating to which module (i.e., features) each model element belongs to. This information supports deriving products from the PLA generated by ModelVars2SPL.

To implement this step, we adopt a graft strategy that enables having a good view of the architecture. This strategy consist in grafting variability information in the merged model by using UML owned comments which are available for each element of a UML class diagram. By adopting this strategy, the obtained PLA can be viewed in any UML editor.

Fig. 8 presents a PLA constructed using the merged model presented in Fig. 7 and the traceability obtained in the first step of ModelVars2SPL. The figure presents the variability information of an attribute of `Bank` with the comment that indicates it belongs to $d^0(+Converter.\{0\})$.

3.6. Additional characteristics

Our approach is designed to provide an automated solution for the three phases of the re-engineering process, but the steps can also be used individually. For instance, if the engineer is only interested in the FM, it is not necessary to execute the last two steps *Model Merging* and *Variability Grafting*. This flexibility on with the entire re-engineering process or with some steps intends to make our approach suitable for different scenarios.

Another point we took into account when designing our approach is the performance for obtaining the solutions. Two steps of ModelVars2SPL, namely *Reverse Engineering of Feature Models* and *Model Merging*, refer to problems where the solutions cannot be easily reached with deterministic algorithms, because of the huge search space available and restrictions to find solutions. In this sense, we employed search-based algorithms that perform better in such scenarios.

4. Evaluation description

This section describes the methodology for the evaluation. Fig. 9 presents an overview of the steps to conduct the evaluation. In the step of *Definition* we follow GQM approach to define research questions and metrics based on our goal. In the *Planning* step we defined algorithms, tools, parameters settings, hardware/platform for the execution, and target applications. In the *Operation*, we run the algorithms/ tools to collect the results and metrics. In the evaluation *Analysis and Interpretation*, we provided analysis of the results and answer the posed research questions.

4.1. Definition

We evaluated ModelVars2SPL[5] according to Wohlin et al. [33] and following the Goal/Question/Metric (GQM) approach [34]. *Our goal is to evaluate ModelsVar2SPL in two aspects: i) capacity to represent all input variants in the PLA and in the FM that are generated, ii) performance*. The questions, goals and metrics are summarized in Table 3. Next we elaborate on them in detail.

[5] The applications, tools, and an illustrative example are publicly available at: https://wesleyklewerton.github.io/reengineering.html.

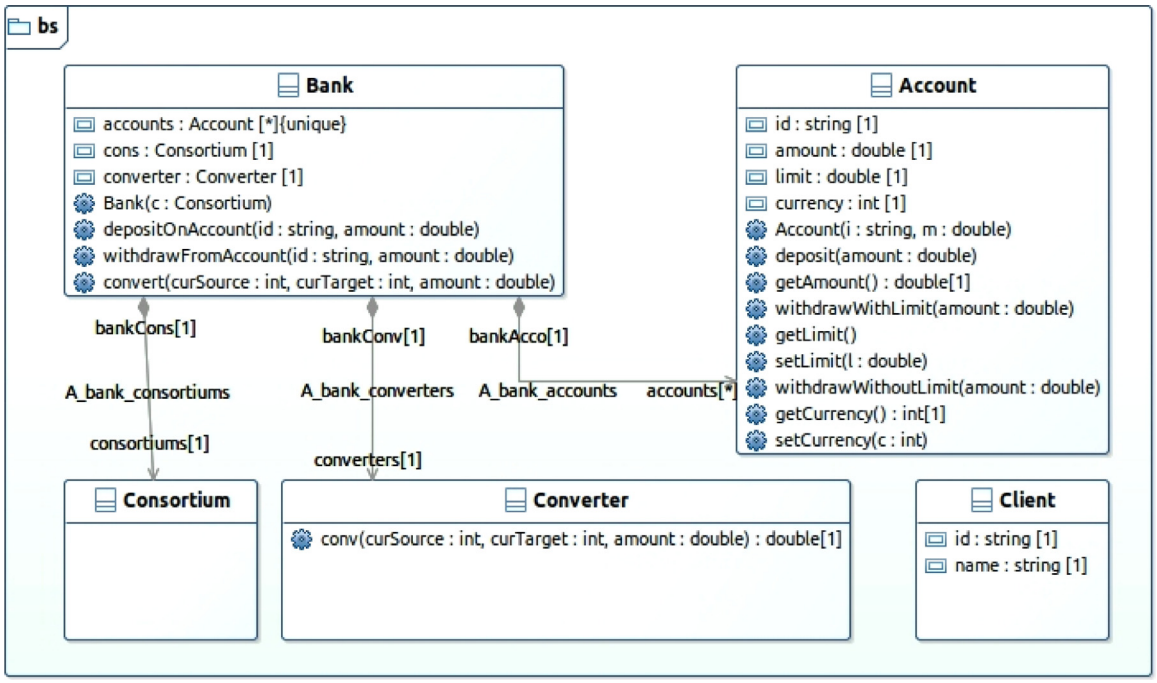


Fig. 7. Example of merged UML class diagram.

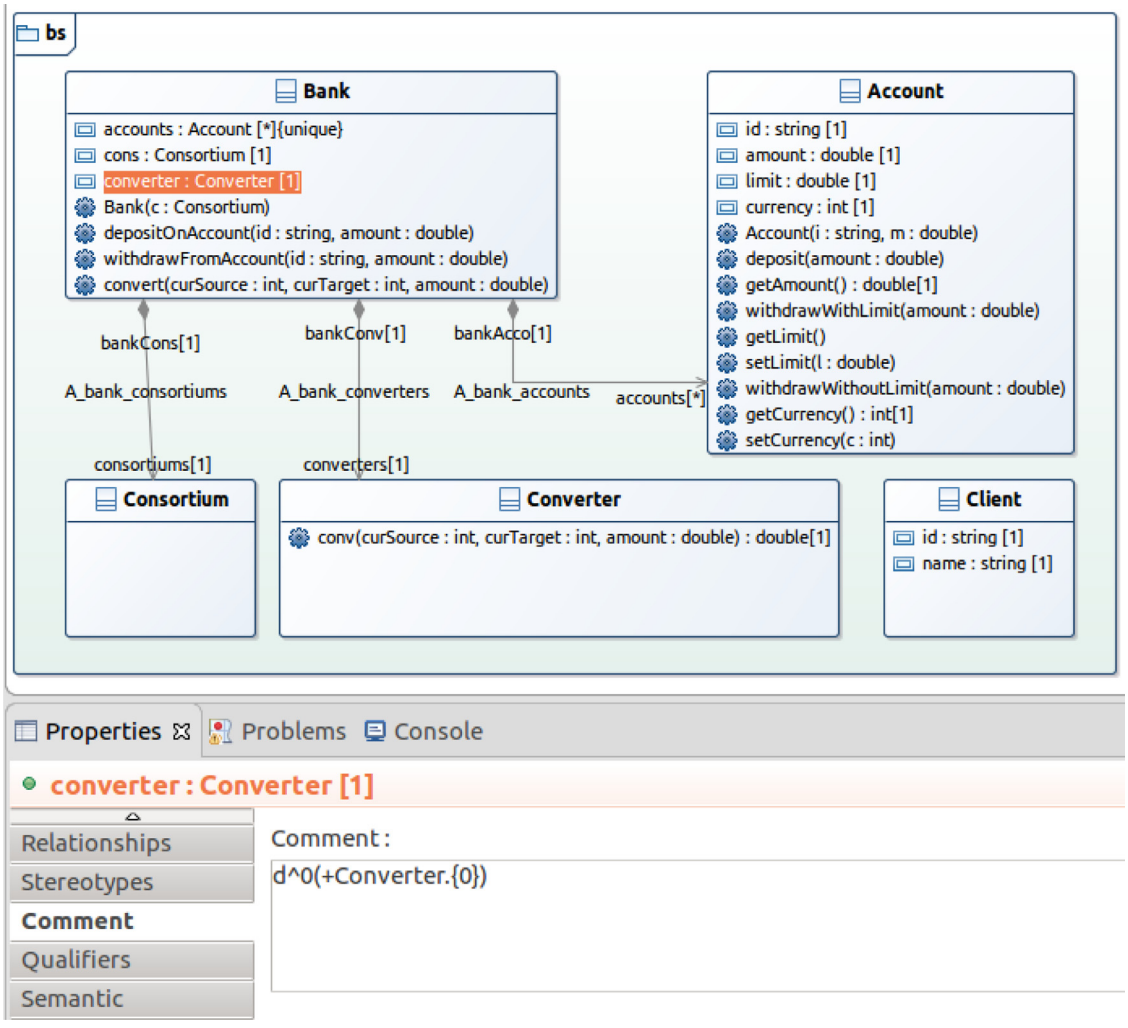


Fig. 8. Example of variabilities in the PLA.

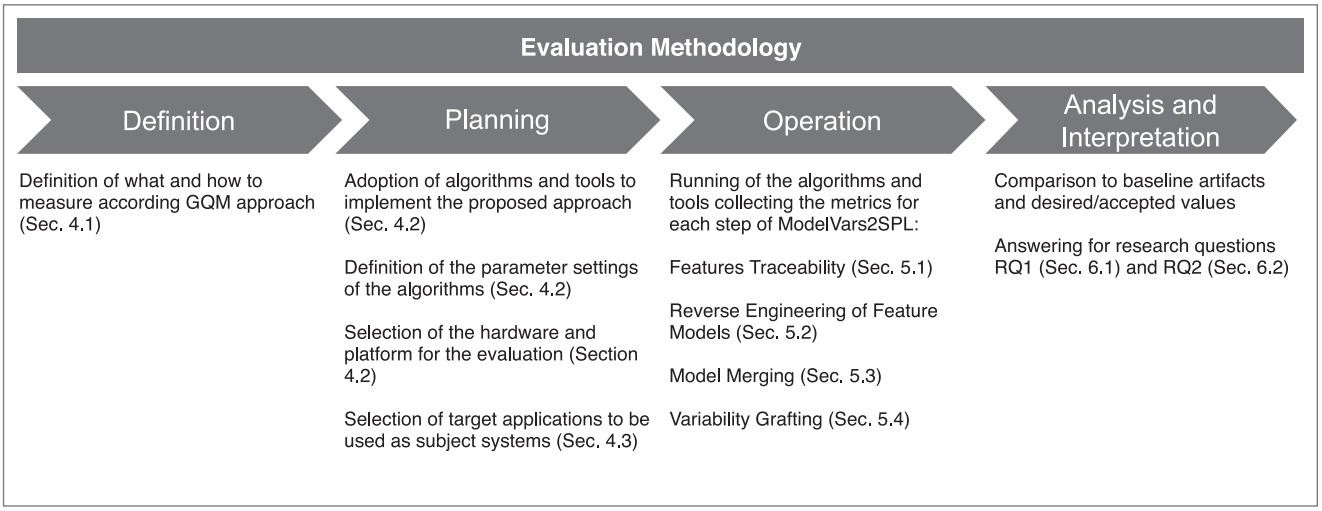


Fig. 9. Steps of the ModelVars2SPL evaluation.

Table 3
Description of our evaluation according to GQM.

Goal	Evaluate the capacity of ModelVars2SPL to represent input variants in PLA and in FM, and its performance
From the point of view	Practitioners
In the context of	SPL extraction from system variants
Question	**Metrics**
RQ1	FMs: Precision (P), Recall (R), and Variability Safety (VS) PLA: Model Similarity (MS)
RQ2	Time to obtain solutions Number of candidate solutions Euclidean Distance from an Ideal Solution (ED)

- *RQ1: How representative are the FMs and PLAs generated by ModelVars2SPL in comparison to the input variants?* ModelVars2SPL is designed to obtain FMs and PLAs that best represent the input variants, intending to keep the behavior of existing products. This research question analyses the features represented in the generated artifacts in relation to the features present in all input variants using the following metrics:
  - *FM Metrics* evaluate how well the generated feature models represent the feature sets of all the input variants. The measures used are Precision (P), Recall (R), and Variability Safety (VS) as presented in Section 3.3.
  - *PLA Metric* evaluates how well the generated PLA represents the variablity expressed in the UML class diagrams of all input variants. The measure used is Model Similarity (MS) as presented in Section 3.4;
- *RQ2: What is the performance of the ModelVars2SPL techniques at each of its four steps?* This research question regards the practical use of the approach. A relevant factor to adopt an approach in practice is how long it takes to obtain solutions and the number of alternative solutions the architects and engineers have to deal with. To this end, we use the following metrics:
  - *Time to obtain solutions:* During the execution of each step of our approach we measured the time performance. For the steps using exact techniques (deterministic algorithms), namely Features Traceability and Variability Grafting, the runtime of each single run for each application was collected. For the steps using search-based techniques (stochastic algorithms), namely Reverse Engineering of FMs and Model Merging, we computed the mean runtime for 30 independent runs;
  - *Number of candidate solutions:* complex software engineering problems usually do not have a single solution. Our approach is designed to reach a set of candidate solutions, since we have a step of multi-objective reverse engineering of FMs. Hence, diverse good solutions might be possible. One of the solutions could be better considering one or more objectives, and other solutions considering the remaining objectives. These solutions are called non-dominated and form the Pareto front. In the evaluation we constructed the Pareto Front based on non-dominated solutions reached considering the 30 independent runs;
  - *Euclidean distance from ideal solution (ED):* ED is a quality indicator that measures how close is a solution from an optimal point (i.e., the ideal solution). An ideal solution has optimal values for each criteria under consideration.

4.2. Implementation details and parameter settings

Our approach deals with models created with the Eclipse Modeling Framework (EMF).[6] EMF is a well-known and widely used set of tools to deal with models [35]. The class diagrams in our approach are represented using EMF-based UML2, which is an implementation of the UML™ 2.x meta-model for the Eclipse platform.

4.2.1. Features Traceability algorithm

For the Feature Traceability step we applied the Linsbauer et al.'s extraction algorithm [30]. This algorithm is available on the ECCO[7] Tool [36]. We developed a parser for ECCO to deal with UML models and decompose class diagrams, allowing the tool to discover the traceability between features and model elements. The parser considers as

[6] https://eclipse.org/modeling/emf/.
[7] https://github.com/llinsbauer/ecco/.

Table 4
NSGA-II parameters.

Parameter	NSGA-II
Number of Fitness Evaluations	200,000
Population Size	200
Crossover Probability	0.7
Feature Tree Mutation Probability	0.5
CTCs Mutation Probability	0.5
Number of Elites	25%
Tournament Size	6
Maximium CTC Percentage for Builder*	0.1
Maximium CTC Percentage for Mutator*	0.5
Independent runs	30

* Relative to number of features.

atomic model elements the class diagram types available in the UML2 EMF-based implementation.[8]

ECCO tool automatically identifies *parent* → *child* dependencies. However, for class diagram relationships we adapted ECCO using our parser for UML2 EMF-based models. We consider in our parser the relationship types:[8] `AssociationImpl`, `UsageImpl`, `DependencyImpl`, `GeneralizationImpl`, `InterfaceRealizationImpl`, `AssociationClassImpl`, `TemplateBindingImpl`. For these relationship types we identify the source and target elements and describe them using the ECCO internal representation. After representing the relationships using ECCO internal types, the dependency graph is automatically generated.

4.2.2. Reverse engineering of FMs algorithm and parameters

In our search-based technique to reverse engineer FMs, the individuals are represented using a simplified version of the SPLX meta-model.[9] This meta-model defines the structure and semantics of FMs [31]. The initial population is composed of random FMs created using the tools FaMa [37] and BeTTy [38]. In this step we applied the Non-Dominated Sorting Genetic Algorithm (NSGA-II) available on the ECJ Framework.[10] The parameters used to configure the algorithm are shown in Table 4 and were chosen based on the literature [39]. They include traditional parameters such as population size (number of solutions), and probabilities to perform the crossover and mutation operations proposed in Section 3.3. The maximum number of fitness evaluations is the stop criterion.

4.2.3. Model merging algorithm and parameters

The implementation of the Model Merging step is based on EMF Diff/Merge. EMF Diff/Merge computes three essential types of differences between models: (i) presence of an unmatched element, which refers to an element in a model that has no match in the opposite model; (ii) presence of an unmatched reference value, which means that a matched element refers another element in only one model; (iii) presence of an unmatched attribute value, where a matched element owns a certain attribute value in only one model. EMF Diff/Merge tool also allows modification of models to incorporate the changes done by the operators.

The search-based technique is implemented using jMetal Framework.[11] We selected the mono-objective generational Genetic Algorithm (gGA). The parameters used to configure gGA are presented in Table 5. In such a table we can see the probabilities for the crossover and mutation operators described in Section 3.4. The maximum number of fitness evaluations is the stop criterion.

[8] From the package: `org.eclipse.uml2.uml.internal.impl.`
[9] http://www.splot-research.org/.
[10] http://cs.gmu.edu/~eclab/projects/ecj/.
[11] http://jmetal.sourceforge.net/.

Table 5
gGA parameters.

Parameter	gGA
Number of Fitness Evaluations	4000
Population Size	200
Crossover Probability	0.95
Mutation Probability	0.2
Number of Elites	2%
Tournament Size	4
Independent runs	30

4.2.4. Variability grafting algorithm

To perform the Variability Grafting step we implemented our own algorithm on top of the ECCO Tool. Our algorithm uses the trace links information discovered by the tool and enriches UML class diagrams with variabilities and features information.

4.2.5. Independent runs

We executed 30 independent runs for each application in the steps Reverse Engineering of FMs and Model Merging [40]. These steps use search-based techniques, which use randomized decisions, and may reach different solutions in each run. The reason of executing many independent runs is to infer the standard behavior of these algorithms. For the steps of Feature Traceability and Variability Grafting, which use deterministic techniques, one single run was executed.

4.2.6. Hardware

The experiments were executed on a machine with an Intel® Core™ i5-3570 CPU with 3.40GHz x 4, 16 GB of memory, and running a 64-bit Linux platform.

4.3. Target applications

For the evaluation we selected ten applications. Each application is a set of different UML class diagram variants where each variant implements different features. The applications are: Banking System (BS), Draw Product Line (DPL), Mobile Media (MM), Video On Demand (VOD), ZipMe (ZM), and Game Of Life (GOL). BS is a small banking application [41]. DPL is a collection of products to draw lines and rectangles [42]. MM is an application to manipulate media files, such as photo, music, and video, on mobile devices [30,43]. For the application MM, we used five versions, namely MMv1 to MMv5. VOD is related to a video-on-demand streaming [30]. ZM is a set of tools for files compression [42]. GOL is a customizable game [42].

Table 6 presents the total number of features, number of optional features, number of variants, mean and standard deviation values of features implemented in each variant, and the mean and standard deviation values of classes, interfaces, attributes, operations, and relationships present in the variants.[12] We reverse engineered the models from Java code using Eclipse MoDisco,[13] except the BS application which was originally a set of UML model variants.

5. Results and analysis

This section presents, for each step of ModelVars2SPL, the results and corresponding analyses.

5.1. Features traceability

The results of the Feature Traceability step are in Table 7. Regarding the module types, the applications MMv1, VOD, GOL, MMv2, MMv3,

[12] To compute the information regarding classes, attributes, operations and relationships we used SDMetrics: http://www.sdmetrics.com.
[13] https://eclipse.org/MoDisco.

Table 6
Detailed Information of the Applications.

Application	Features		Variants	Mean (Std Deviation) Features per Variant	Mean (Std Deviation) per variant				
	Total	Mandatory			Classes	Interfaces	Attributes	Operations	Relationships
BS	4	1	8	2.50 (0.93)	4.00 (0.76)	0.00 (0.00)	7.00 (1.31)	10.00 (2.93)	2.88 (1.36)
DPL	6	1	16	3.75 (1.13)	4.69 (1.25)	0.00 (0.00)	13.50 (5.39)	35.44 (10.25)	5.13 (1.63)
MMv1	5	2	3	3.00 (0.00)	23.00 (0.00)	1.00 (0.00)	22.00 (0.00)	121.00 (0.00)	13.00 (0.00)
VOD	11	6	32	8.50 (1.14)	37.00 (3.05)	0.00 (0.00)	280.00 (0.00)	233.00 (9.34)	91.50 (7.89)
ZM	7	2	32	4.50 (1.14)	25.00 (1.76)	3.00 (0.00)	193.00 (6.10)	283.50 (23.54)	74.75 (5.83)
GOL	15	3	28	6.93 (1.46)	15.61 (1.26)	1.96 (0.19)	22.79 (2.78)	74.57 (5.07)	32.32 (1.31)
MMv2	6	2	4	3.25 (0.50)	24.00 (0.00)	1.00 (0.00)	25.25 (0.50)	134.25 (2.50)	21.00 (0.00)
MMv3	7	2	9	3.67 (0.50)	24.00 (0.00)	1.00 (0.00)	25.67 (0.50)	135.67 (2.18)	21.00 (0.00)
MMv4	8	2	12	3.75 (0.45)	28.25 (0.45)	1.00 (0.00)	25.75 (0.45)	150.00 (1.95)	25.75 (1.36)
MMv5	9	2	33	4.45 (0.67)	31.18 (3.72)	1.00 (0.00)	32.55 (8.43)	167.64 (19.37)	27.45 (2.05)

Table 7
Feature traceability results.

Application	Module Type		Module Dependencies	Runtime	
	Base	Derivative		sec	ms
BS	4	3	6	1	987
DPL	7	4	10	2	206
MMv1	1	0	0	2	58
VOD	5	0	4	13	570
ZM	6	20	28	17	394
GOL	9	0	14	12	373
MMv2	2	0	1	2	230
MMv3	3	0	2	2	930
MMv4	6	0	5	3	723
MMv5	11	0	12	6	875

Table 8
Reverse engineering of FMs results.

Application	Pareto Front	Best ED	Best ED Solution			Runtime		
			P	R	VS	m	s	ms
BS	1	0.0	1.0	1.0	1.0			234
DPL	1	0.0	1.0	1.0	1.0	1	0	252
MMv1	1	0.0	1.0	1.0	1.0		5	180
VOD	1	0.0	1.0	1.0	1.0			746
ZM	1	0.0	1.0	1.0	1.0		1	395
GOL	8	0.357	1.0	0.643	1.0	5	37	642
MMv2	1	0.0	1.0	1.0	1.0		3	740
MMv3	1	0.0	1.0	1.0	1.0		4	820
MMv4	91	0.101	1.0	1.0	0.899	5	9	975
MMv5	261	0.184	1.0	0.818	0.972	39	49	388

ED = Euclidean Distance, P = Precision, R = Recall, VS = Variability Safety.

MMv4 and MMv5 are composed of only base modules, which means there are no implementations of feature interactions. On the other hand, the applications BS and DPL have some derivative modules and ZM has many derivative modules in comparison to the number of base modules. Consider Table 6 which presents the applications information. We can see that ZM is one of the largest applications and has the largest proportion of relationships in comparison to the other model element types. This characteristic may justify the high number of derivative modules of ZM. This also may be related to the domain characteristics that demand features to interact in order to implement the functionalities.

Taking into account the total number of modules, MMv1 is composed of only one module. Despite of having five features (Table 6), the three variant models of MMv1 are similar, because the variability happens in a granularity level (statements inside operations) not represented in the UML class diagrams. MMv5 has the largest number of base modules and is the application that has the largest number of variants.

The module dependencies represent the relationships between modules. All applications have module dependencies, except MMv1. As expected, the number of dependencies is related to the number of modules, as we can observe analyzing the second and third columns of Table 7 in comparison to the fourth column. Besides, we can see that ZM, GOL and MMv5 are the applications with the largest number of module dependencies. ZM, GOL and MMv5 are also the applications with the largest number of relationships and operations in comparison to the other model elements (see Table 6). This led us to infer that relationships and operations are related with module dependencies.

The runtime seems to be more related to the number of relationships in the variants (Table 6). See applications VOD, ZM, and GOL. For them, ModelVars2SPL had the largest runtimes. These cases also have the largest mean number of relationships in the variants. On the other hand, for MM5, an application with the largest number of features and variants, the runtime is smaller than the former three applications.

In summary, we can see that ModelVars2SPL performs well on identifying base and derivative modules, and module dependencies. The time required for the feature traceability increases with to the number of features and variants. But even for the largest applications, the necessary runtime is only few seconds.

5.2. Reverse engineering of feature models

With respect to the step of Reverse Engineering of FMs, Table 8 presents the obtained results. For seven applications the algorithm NSGA-II was able to reach optimal solutions (P = 1.0, R = 1.0, VS = 1.0), leading to ED values equal to 0.0. A value equal to 1.0 for precision means that all configurations of the input variants (i.e., features set) are denoted by the reverse engineered FM; 1.0 for recall means the input products are exactly represented, there are no surplus configurations denoted by the FM; and the best value for variability safety (VS = 1.0) means that there are no violated dependencies in the FM regarding the dependency graph. For the applications GOL, MMv4 and MMv5, the algorithm NSGA-II obtained a set of solutions, indicating a conflict among the values of the objectives being optimized. The cases with conflicting objectives take the largest amount of time. For instance, NSGA-II took almost 40 minutes to complete the evolutionary process for the application MMv5. Considering the characteristics of these three applications, we can see they all have few mandatory features in relation to the number of total ones, as seen in the second and third columns of Table 6. Furthermore, the variants of these applications implement less than 50% of features in each variant on average (fifth column of Table 6). For such cases the search space, and consequently the number of solutions found, is greater.

Taking into account the Pareto Fronts with cardinality greater than one, Fig. 10 presents the solutions in the search space. The graphs are presented by considering pairs of objectives, for an easy visualization. In the application GOL, the conflicting objectives are related to P and R (Fig. 10(a)), since the values of VS in graphs of Fig. 10(b) and 10(c) are equal to 1. NSGA-II explored well the search space of GOL, as we can observe in Fig. 10(a), the solutions cover the entire search space.

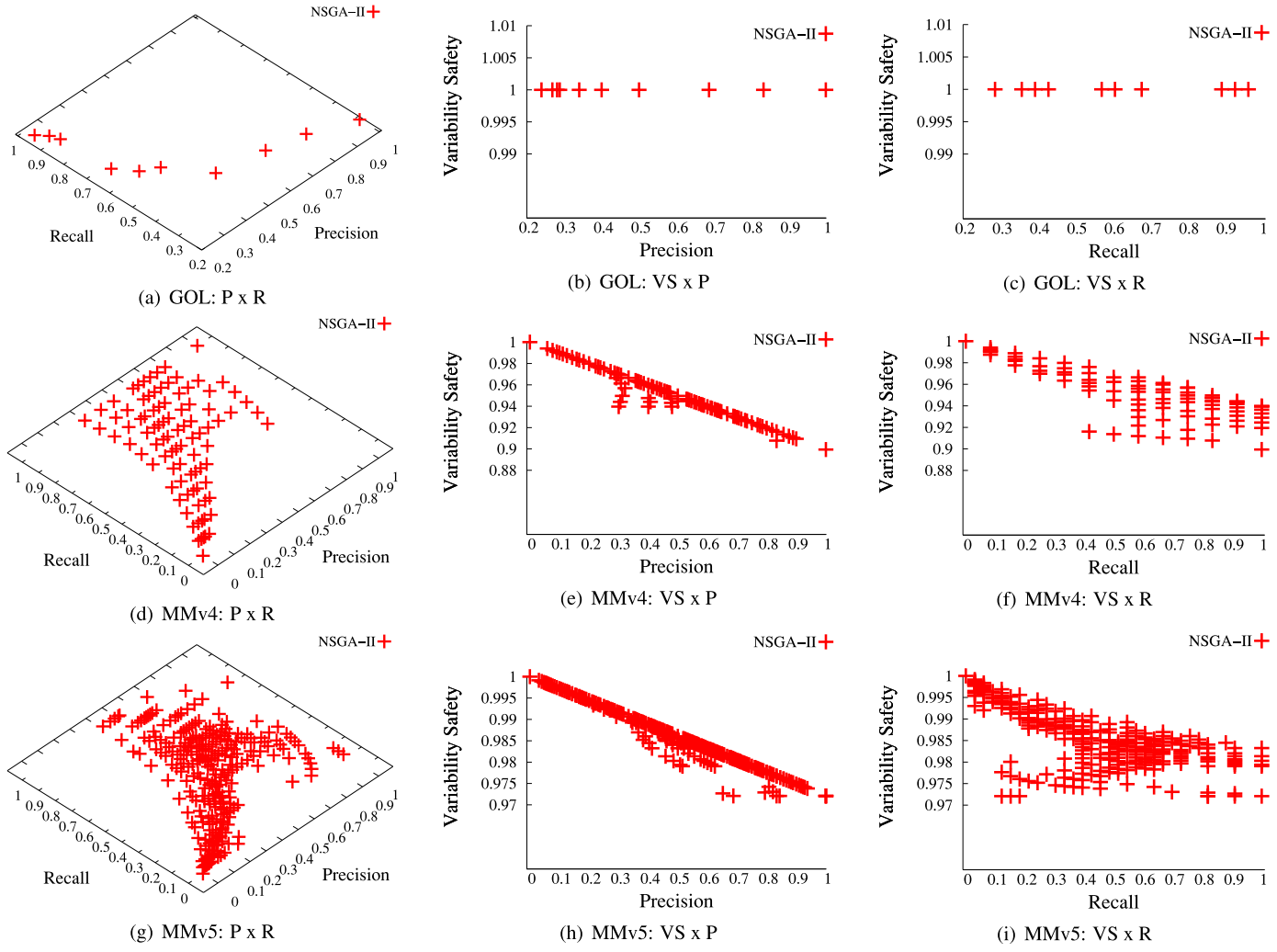


Fig. 10. Solutions on the search space for reverse engineering feature models.

Table 9
Model merging results.

Application	Mean of the Best*	Best Solution*	Elements					Runtime			
			Cl	Int	Attr	Op	Rel	h	m	s	ms
BS	20.07	20	5	0	9	15	5		1	24	724
DPL	60.53	51	5	0	21	41	5		9	6	269
MMv1	0.00	0	23	1	22	121	13		7	11	326
VOD	324.27	302	42	0	282	249	186	2	11	2	673
ZM	348.27	264	28	3	204	386	94	2	18	19	501
GOL	163.63	146	19	2	33	93	39		34	31	417
MMv2	0.00	0	24	1	26	138	21		10	5	797
MMv3	0.00	0	24	1	28	141	21		20	0	752
MMv4	9.00	9	29	1	29	158	27		28	50	764
MMv5	348.27	264	36	1	46	200	30	1	23	49	51

* Model Similarity, Cl = number of classes, Int = number of interfaces, Attr = number of attributes, Op = number of operations, Rel = number of relations.

Regarding applications MMv4 and MMv5, there are inter-dependencies among the three objectives, but in the same way, NSGA-II explored well the search space. Based on the graphs we can see the solutions are spread over the search space.

5.3. Model merging

Table 9 presents the results obtained in this step. The mean value of the best merged model (second column) considers the solutions found in all 30 independent runs. The best solution among the 30 independent runs is presented in third column.

From the fitness values of the best solutions we can observe that the most complete model obtained is not similar to any existing variant, because of the number of differences from the input variants, except for MMv1, MMv2, and MMv3 where the model variants are very similar. For instance, the merged model obtained for the application VOD has 302 differences from the 32 input variants (Table 6). This means that even merging many existing differences, a model completely similar to

Table 10
Variability grafting results.

Application	Model elements Annotated	Runtime	
		s	ms
BS	44	1	481
DPL	103	1	811
MMv1	306	2	806
VOD	728	8	972
ZM	857	9	31
GOL	230	4	524
MMv2	327	2	899
MMv3	333	2	927
MMv4	355	3	204
MMv5	448	3	958

all input variants is difficult to be obtained. This situation justifies the use of search-based approaches. Analyzing the mean fitness of the best solutions we can see that the best solution is not always reached. This happens mainly because of the huge search space, and the entire exploration of the search space is computationally infeasible. This is observed by looking at the runtimes. For two applications, namely VOD and ZM, each independent run took more than two hours on average.

5.4. Variability grafting

The results of the Variability Grafting are presented in Table 10. The number of model elements annotated with the traceability information is presented in the second column. The runtime for each application is in the last column. The applications VOD and ZM have the largest models, therefore the variability grafting algorithm took the largest runtime. However, the runtime did not take more than ten seconds.

6. Discussion of the results

This section discusses the results, by providing answers to our research questions (Section 4), and discussing possible threats to validity.

6.1. Answering RQ1

To answer this question, we summarize some results of the obtained FMs and PLAs in Table 11. The basis for the comparison is the number of feature sets used as input, presented in the second column of the table, and the number of model elements of the baseline class diagram (third column). The baseline of each application is the most complete UML class diagram variant used as input (i.e., that implements the greatest number of features).

To answer RQ1, the first point to be taken into account is the instantiation of the same input variants as products of the SPL. Initially we consider the obtained FM for each application. Observing the feature sets used as input (second column) and the feature sets present in the obtained FMs (fourth column) of Table 11, we can see that our approach is able to represent exactly the same input configurations of products for eight out of ten applications. Despite of not finding optimal solutions for two applications, the obtained FMs have more present feature sets than absent ones (fifth column). For instance, in the GOL application the obtained FM denotes 18 input feature sets, and only ten feature sets are absent. This happens for the applications GOL and MMv5, which have the largest number of features and variants respectively. With respect to the violated dependencies, only in two applications the obtained FM denotes feature sets that do not satisfy some dependencies of the dependency graph, namely MMv4 and MMv5. These two applications have only two optional features and the configurations of products have less than 50% of features on average (see Table 6). This means that there is not much available information for the evolutionary process of NSGA-II. In spite of that, we can say our approach is able to generate variability-safe FMs, or in other words, our approach is consistent to the design artifacts. Thus, we claim that for the great majority of the cases our approach finds solutions that represent well the input variants.

Regarding the PLAs, we analyze the baseline in comparison to the PLAs obtained with our approach. In Table 11 the number of model elements in the PLA (seventh column) is greater than the model elements in the baseline (third column) except for MMv4. A greater number of model elements means that the PLA is more complete than the baseline. To corroborate our findings, Fig. 11 presents the graphs with the number of model elements. We can observe that the number of model elements in the PLA is very similar to the number in the baseline in the applications BS, DPL and MMV1, the smallest applications. For the applications GOL, VOD and ZM, the PLA has more model elements than the baseline, which is expected.

In the last two columns of Table 11 we can see the number of differences between the baseline and PLA taking into account the two directions of comparison. For instance, considering the BS application, the baseline has 16 different model elements in comparison to the PLA. On the other hand, the PLA has only 3 differences in comparison to the baseline. Analyzing these differences, we can see in all applications that the PLA is more similar to the baseline than the other way around.

6.1.1. RQ1 discussion

Based on the results presented above, we can evaluate how representative the solutions are, regarding the input variants. We can attest that the FMs generated with ModelVars2SPL in most of the cases represent the exact input variants, since most of the times (8 out of 10) the FMs represent exactly the desired input feature sets. Also, only in two applications (2 out of 10) the FMs have violated dependencies. In general, ModelVars2SPL can reach solutions in accordance to the design artifacts. In all applications the PLA is similar or more complete than the baselines, indicating our approach is able to merge the input UML class

Table 11
Results for the obtained FMs and PLAs.

Application	Input Feature Sets	Baseline Model Elements*	Obtained FM			Obtained PLA		
			Present Feature Sets	Absent Feature Sets	Violated Dependencies	Model Elements*	Differences Baseline>PLA	Differences PLA>Baseline
BS	8	32	8	0	0	34	16	3
DPL	16	72	16	0	0	72	303	2
MMv1	3	180	3	0	0	180	827	0
VOD	32	675	32	0	0	759	2122	36
ZM	32	656	32	0	0	715	2588	99
GOL	28	178	18	10	0	186	1954	521
MMv2	4	210	4	0	0	210	953	0
MMv3	9	214	9	0	0	215	1989	712
MMv4	12	245	12	0	12	244	2195	819
MMv5	33	313	27	6	27	313	2788	1038

* (Classes + Interfaces + Attributes + Operations + Relationships).

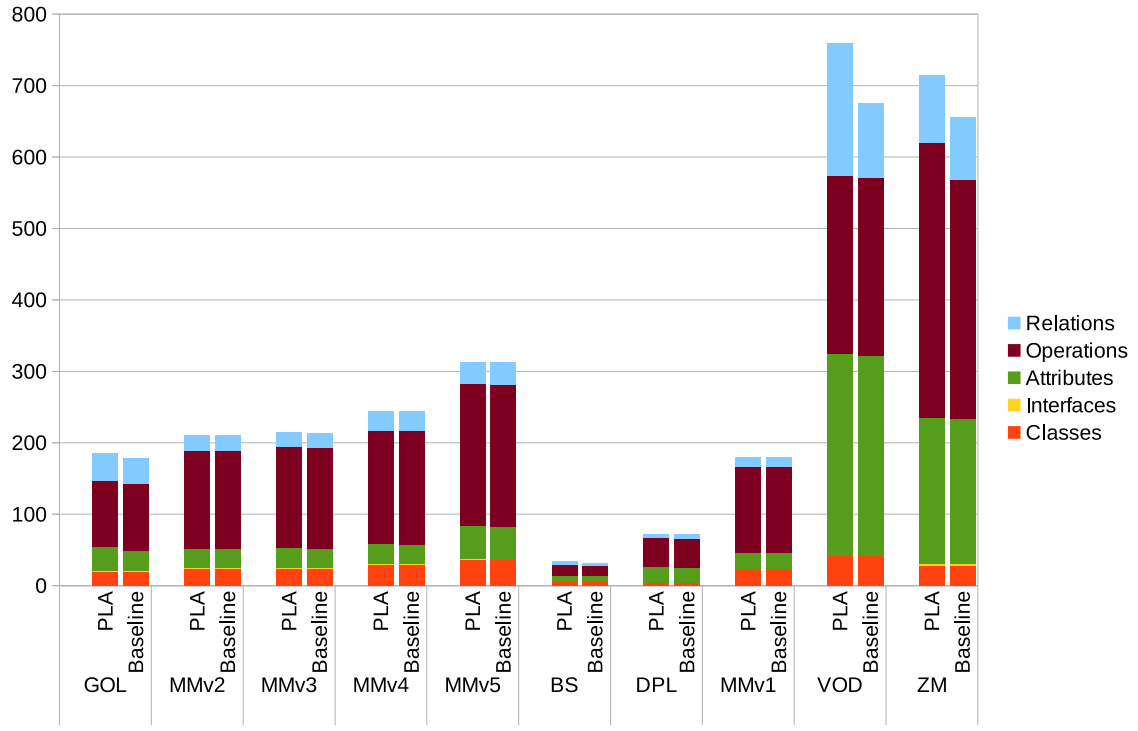


Fig. 11. Baselines and PLAs.

Table 12
Approach runtime.

Application	Feature Traceability		Reverse Engineer of FMs			Model Merging				Variability Grafting		Total Runtime			
	sec	ms	min	sec	ms	hour	min	sec	ms	sec	ms	hour	min	sec	ms
BS	1	987			234		1	24	724	1	481		1	28	426
DPL	2	206	1	0	252		9	6	269	1	811		10	10	538
MMv1	2	58		5	180		7	11	326	2	806		7	21	370
VOD	13	570			746	2	11	2	673	8	972	2	11	25	962
ZM	17	394		1	395	2	18	19	501	9	31	2	18	47	322
GOL	12	373	5	37	642		34	31	417	4	524		40	25	956
MMv2	2	230		3	740		10	5	797	2	899		10	14	667
MMv3	2	930		4	820		20	0	752	2	927		20	11	429
MMv4	3	723	5	9	975		28	50	764	3	204		34	7	667
MMv5	6	875	39	49	388	1	23	49	51	3	958	2	3	49	272

diagram variants properly. In summary, we can say that the solutions represent well the input variants.

6.2. Answering RQ2

To answer RQ2 we have to consider the solutions of all steps. In summary, the step Features Traceability obtains solutions as expected, discovering properly the base and derivative modules, and dependencies (Table 7). The FMs reverse engineered in the second step describe well the input feature sets, as shown by the values of P, R, and VS. The majority of the solutions found with the search-based technique are optimal solutions in the Pareto front with the best ED values (Table 8). The evolutionary process in the step of Model Merging is able to reach good global UML class diagrams by exploring the search space. As previously discussed, the solutions are very similar to the baseline variants (Table 9). Finally, we observed that our algorithm in the Variability Grafting step can successfully include annotations in model elements to generate good PLAs (Table 10).

Another aspect of this question is the runtime needed to obtain a solution. In the previous section, we presented the runtime of each step individually, but here we consider it entirely. Table 12 presents the runtime of the four steps of the approach and the total runtime. The runtime of the reverse engineering of FMs is related to the input information as shown in Table 6, i.e. a lower runtime is observed when an application has few optional features (low variability). This low variability implies simpler FMs which are easier to reach. However, the bottleneck regarding the runtime is the step of model merging. As we can see in Table 12, for the applications VOD, ZM, and MMv5 ModelVars2SPL took more than 2 hours in this step. On closer inspection, these applications have larger number of variants and model elements (see Table 6), fact that cause the computation of the fitness functions (based on model diffing) to take longer. Even when considering the most complex applications, the solutions are obtained with acceptable runtime from a practitioner's point of view. For example, conducting the re-engineering process manually would certainly require more time and would be error prone.

6.2.1. RQ2 discussion

Considering the artifacts obtained in each step we can attest that ModelVars2SPL can successfully aid the extraction of SPL core design artifacts. The good representativeness of the FMs and PLAs obtained, as

discussed in Section 5.1, is a consequence of the good results obtained throughout all the steps in ModelVars2SPL. Furthermore, the runtime of the steps allows the approach to be applied in practice, thus providing developers a sound FM and a PLA that serves as launchpad for the re-engineering process.

6.3. Threats to validity

A possible threat to validity of our results is the parameter settings for the search-based techniques. To reduce such a threat we used parameter values recommended in the literature. But a tuning phase can lead to even better results.

The second threat is the influence of the applications. Despite of using only ten applications, they are systems from different domains, and have different sizes in terms of variants, features, and model elements. Based on that, we argue that these applications can provide evidence about the soundness of our approach. But nonetheless further studies should be conducted in the future.

The third threat concerns the comparison to other approaches. Given the lack of approaches with the same goal of ModelVars2SPL, we compared the models generated by our approach with models containing all features present in the systems, assuming that these models are the closest to an ideal solution.

The fourth threat to validity is the scalability of our approach. We did not perform an in-depth analysis regarding the performance of our approach controlling the size of the input and observing until what point the solutions remain good. This was so because of the lack of publicly available applications. However, we tried to mitigate this threat by using applications with different sizes in terms of number of variants, features, and model elements. We can argue that for larger input set of variants, or for very large models, it is possible to fragment the input in some subsets and then use these different subsets as input. In the end, the solutions of the subsets could be used as input for a new execution of ModelVars2SPL. This will be evaluated in future experiments.

The last threat refers to the validation of the solution by practitioners. We intend to evaluate this in future work by: i) comparing the models generated by them with the models generated by our approach; and ii) analysing what generated models they select in order to study the influence of the domain knowledge in the choice of different models, since there are no canonical representations of FMs and PLAs. This threat was mitigated by considering real case studies that have been extensively studied in the literature of the SPL community.

7. ModelVars2SPL practical usage

Next we discuss the usefulness of the artifacts generated by ModelVars2SPL, showing the approach's benefits for practitioners.

7.1. Communication between practitioners and stakeholders

On one hand, the FMs generated by ModelVars2SPL can support the communication between development team and stakeholders, allowing better managerial decision regarding the migration of existing variants to an SPL. On the other hand, the PLAs can aid technical decisions, since they allow a better understanding of the implementation, presenting commonalities and variabilities.

7.2. Re-engineering planning

The FMs and PLAs can be used as a source of information to support practitioners on deciding a sequence of products to be migrated to an SPL. Based on these artifacts, the practitioners can identify variant outliers, i.e. those variants that became too different from their familiars, and remove these variants from the re-engineering process. Furthermore FMs and PLAs allow the estimation of financial and human resources to perform the re-engineering process, supporting managerial decisions.

7.3. Maintenance of the variants

The core design assets generated by ModelVars2SPL can be used as a source of information for bug fixing, providing a global view of the whole system family, easing the identification of the parts to be maintained/fixed. In addition, the ModelVars2SPL solutions can help in the identification of bad smells, since the PLA provides a global implementation view of features that are spread across different variants easing their analysis.

7.4. System evolution

FMs and PLAs can also aid the evolution of an existing system. From the FM, practitioners can reason and plan the release of new product with new configuration of features, decide the development of a new functionality to fulfill domain gaps to be more competitive. The PLA can support the platform migration of existing products, e.g. from desktop to mobile, cloud, or micro-services.

7.5. Better test cases

By using PLAs generated by ModelVars2SPL, software engineers and testers can observe the interaction among different parts that implement a feature. In collaboration with the FM, interactions among features can also be taken into account. In this way the practitioners can design better test cases to exercise such different parts.

8. Concluding remarks

This work presents the ModelVars2SPL approach for extraction of two main SPL core design assets - FMs and PLAs. ModelVars2SPL is composed of four steps covering all three phases of the re-engineering process from system variants to SPLs. ModelVars2SPL takes as input a set of UML class diagram for each variant and the set of features each variant implements. The generated FM describes the configuration of the variants and the produced PLA represents the model elements related to each feature.

An advantage and distinguishing characteristic of ModelVars2SPL is that it uses UML design models. Thus, it is independent of programming languages and supports the re-engineering process at the design level further allowing developers to have a broader view of the SPL beyond code. In addition, the SPL core design artifacts are obtained automatically. ModelVars2SPL uses search-based techniques to better solve the complex activities as the identification of domain constraints. In this way, ModelVars2SPL can reduce effort to extract SPLs from system variants and make the adoption of SPL using an extractive approach more attractive, since most organizations commonly have the artifacts required as input for ModelVars2SPL.

We evaluated ModelVars2SPL with ten applications from different domains and complexities. The evaluation focused on the quality of the generated FMs and PLAs, for which we considered how well they represent the input variants and the runtime performance to obtain candidate solutions. The results indicate that ModelVars2SPL is effective to obtain FMs and PLAs that represent the input variants and it does so within an acceptable runtime. For the former, our approach reached FMs that represent the exact set of features sets used as input, and the constructed PLAs are similar to the baselines. For the latter, in the worst scenario all the steps of ModelVars2SPL took less than 2 h and 30 min.

Concerning the benefits of ModelVars2SPL, in terms of practical use of its generated FMs and PLAs, we argued that they can be used to aid the communication between practitioners and stakeholders, the re-engineering planning, maintenance of existing products, to support system evolution for development of new products and platform migration, and to improve quality by easing derivation of better test cases.

For future work we can mention the use of the PLA for automatically instantiating products. The approach can be extended to deal with

meta-models[14] that represent a wide set of models, besides class diagrams. Other extension is the use of search algorithms based on the user preferences, which could be incorporated in the approach to capture the developer's needs. For example, there are situations where practitioners prefer to keep semantics in the models instead of having an improved structure of the model. Experiments should be conducted to better evaluate ModelVars2SPL, with other applications and scenarios.

In addition, we plan to explore the use of ModelVars2SPL with other goals. One example is its use for reconciling different models created by different stakeholders. During the design phase, different stakeholders can model similar parts of a system, which might be conflicting. ModelVars2SPL can be used to merge these models and provide an unified vision of the system. Another research opportunity is to investigate the use of ModelVars2SPL to support activities of bug fixing, detection of bad smells and refactoring, evolution with new products, improvement of the performance of the products, optimization of the variability management, and test case design.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the Brazilian Agencies CAPES: 7126/2014-00 and CNPq: 453678/2014-9, 305358/2012-0 and 408356/2018-9. Natural Sciences and Engineering Research Council of Canada RGPIN-2017-05421.

References

[1] C.W. Krueger, Software reuse, ACM Comput. Surv. 24 (2) (1992) 131–183, doi:10.1145/130844.130856.
[2] R. Holmes, R.J. Walker, Systematizing pragmatic software reuse, ACM Trans. Softw. Eng.Methodol. 21 (4) (2013) 20:1–20:44, doi:10.1145/2377656.2377657.
[3] N. Kulkarni, V. Varma, Perils of opportunistically reusing software module, Software 47 (7) (2017) 971–984, doi:10.1002/spe.2439.
[4] D. Faust, C. Verhoef, Software product line migration and deployment, Software: Practice and Experience 33 (10) (2003) 933–955.
[5] K. Pohl, G. Böckle, F.J.v.d. Linden, Software Product Line Engineering: Foundations, Principles and Techniques, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
[6] P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
[7] F.J.v.d. Linden, K. Schmid, E. Rommes, Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.
[8] M.A. Laguna, Y. Crespo, A systematic mapping study on software product line evolution: from legacy system reengineering to product line refactoring, Sci. Comput. Programm. 78 (8) (2013) 1010–1034 Special section on software evolution, adaptability, and maintenance & Special section on the Brazilian Symposium on Programming Languages. doi:10.1016/j.scico.2012.05.003.
[9] W.K.G. Assunção, R.E. Lopez-Herrejon, L. Linsbauer, S.R. Vergilio, A. Egyed, Reengineering legacy applications into software product lines: a systematic mapping, Empir. Softw. Eng. 22 (6) (2017) 2972–3016, doi:10.1007/s10664-017-9499-z.
[10] G. Canfora, M. Di Penta, L. Cerulo, Achievements and challenges in software reverse engineering, Commun. ACM 54 (4) (2011) 142–151.
[11] M. Hasbi, E.K. Budiardjo, W.C. Wibowo, Reverse engineering in software product line - a systematic literature review, in: 2Nd International Conference on Computer Science and Artificial Intelligence, in: CSAI '18, ACM, New York, NY, USA, 2018, pp. 174–179.
[12] J. Martinez, W.K.G. Assunção, T. Ziadi, Espla: a catalog of extractive SPL adoption case studies, in: Proceedings of the 21st International Systems and Software Product Line Conference - Volume B, in: SPLC '17, ACM, New York, NY, USA, 2017, pp. 38–41, doi:10.1145/3109729.3109748.
[13] J. Martinez, T. Ziadi, J. Klein, Y. le Traon, Identifying and visualising commonality and variability in model variants, in: J. Cabot, J. Rubin (Eds.), Modelling Foundations and Applications, Springer International Publishing, Cham, 2014, pp. 117–131.
[14] D. Wille, S. Schulze, I. Schaefer, Variability mining of state charts, in: 7th International Workshop on Feature-Oriented Software Development, in: FOSD'16, ACM, New York, NY, USA, 2016, pp. 63–73, doi:10.1145/3001867.3001875.
[15] J. Rubin, M. Chechik, Combining related products into product lines, in: J. de Lara, A. Zisman (Eds.), Fundamental Approaches to Software Engineering, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 285–300.
[16] T. Ziadi, L. Frias, M.A.A. da Silva, M. Ziane, Feature identification from the source code of product variants, in: 16th European Conference on Software Maintenance and Reengineering, 2012, pp. 417–422, doi:10.1109/CSMR.2012.52.
[17] M. Mefteh, N. Bouassida, H. Ben-Abdallah, Implementation and evaluation of an approach for extracting feature models from documented UML use case diagrams, in: 30th Annual ACM Symposium on Applied Computing, in: SAC'15, ACM, New York, NY, USA, 2015, pp. 1602–1609, doi:10.1145/2695664.2695907.
[18] J. Bayer, T. Forster, D. Ganesan, J.-F. Girard, I. John, J. Knodel, R. Kolb, D. Muthig, Definition of Reference Architectures Based on Existing Systems, Technical Report Report No. 034.04/E, Fraunhofer IESE-Report No. 034.04/E, 2004.
[19] K. Kumaki, R. Tsuchiya, H. Washizaki, Y. Fukazawa, Supporting commonality and variability analysis of requirements and structural models, in: 16th International Software Product Line Conference, in: SPLC, ACM, 2012, pp. 115–118, doi:10.1145/2364412.2364431.
[20] J. Martinez, T. Ziadi, T.F. Bissyandé, J. Klein, Y. le Traon, Automating the extraction of model-based software product lines from model variants, in: 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015, pp. 396–406.
[21] T. Berger, S. She, R. Lotufo, A. Wasowski, K. Czarnecki, A study of variability models and languages in the systems software domain, IEEE Trans. Softw. Eng. 39 (12) (2013) 1611–1640, doi:10.1109/TSE.2013.34.
[22] I. Reinhartz-Berger, M. Kemelman, Extracting core requirements for software product lines, Requir. Eng. (2019), doi:10.1007/s00766-018-0307-0.
[23] S. She, R. Lotufo, T. Berger, A. Wąsowski, K. Czarnecki, Reverse engineering feature models, in: 33rd International Conference on Software Engineering, in: ICSE'11, ACM, New York, NY, USA, 2011, pp. 461–470, doi:10.1145/1985793.1985856.
[24] M. Harman, S.A. Mansouri, Y. Zhang, Search-based software engineering: trends, techniques and applications, ACM Comput. Surv. 45 (1) (2012) 11:1–11:61, doi:10.1145/2379776.2379787.
[25] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, A. Tang, What industry needs from architectural languages: asurvey, IEEE Trans. Softw. Eng. 39 (6) (2013) 869–891, doi:10.1109/TSE.2012.74.
[26] P. Lago, I. Malavolta, H. Muccini, P. Pelliccione, A. Tang, The road ahead for architectural languages, IEEE Softw. 32 (1) (2015) 98–105, doi:10.1109/MS.2014.28.
[27] J. Rubin, M. Chechik, From products to product lines using model matching and refactoring, in: 2nd International Workshop on Model-driven Approaches in Software Product Line Engineering (MAPLE '10), collocated with the 14th International Software Product Line Conference (SPLC '10), 2010.
[28] J. Rubin, Cloned Product Variants: From Ad-hoc to Well-managed Software Reuse, University of Toronto, Graduate Department of Computer Science, 2014 Ph.D. Thesis.
[29] J. Rubin, M. Chechik, N-way model merging, in: 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), ACM, 2013, pp. 301–311, doi:10.1145/2491411.2491446.
[30] L. Linsbauer, E.R. Lopez-Herrejon, A. Egyed, Recovering traceability between features and code in product variants, in: 17th International Software Product Line Conference, in: SPLC 2013, ACM, New York, NY, USA, 2013, pp. 131–140, doi:10.1145/2491627.2491630.
[31] W.K.G. Assunção, R.E. Lopez-Herrejon, L. Linsbauer, S.R. Vergilio, A. Egyed, Multi-objective reverse engineering of variability-safe feature models based on code dependencies of system variants, Empir. Softw. Eng. (2016) 1–32, doi:10.1007/s10664-016-9462-4.
[32] W.K.G. Assunção, S.R. Vergilio, R.E. Lopez-Herrejon, Discovering software architectures with search-based merge of UML model variants, in: G. Botterweck, C. Werner (Eds.), Mastering Scale and Complexity in Software Reuse: 16th International Conference on Software Reuse (ICSR), Springer, 2017, pp. 95–111.
[33] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering: An Introduction, Kluwer Academic Publishers, Norwell, MA, USA, 2000.
[34] V.R. Basili, G. Caldiera, H.D. Rombach, The goal question metric approach, Encycl. Softw. Eng. 2 (1994) (1994) 528–532.
[35] D. Steinberg, F. Budinsky, E. Merks, M. Paternostro, EMF: Eclipse Modeling Framework, Pearson Education, 2008.
[36] S. Fischer, L. Linsbauer, R.E. Lopez-Herrejon, A. Egyed, The ECCO tool: extraction and composition for clone-and-own, in: 37th International Conference on Software Engineering - Volume 2, in: ICSE'15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 665–668.
[37] D. Benavides, S. Segura, P. Trinidad, A.R. Cortés, FaMa: tooling a framework for the automated analysis of feature models, in: International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS), 2007, pp. 129–134.
[38] S. Segura, J. Galindo, D. Benavides, J.A. Parejo, A.R. Cortés, BeTTy: benchmarking and testing on the automated analysis of feature models, in: International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS), 2012, pp. 63–71.
[39] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, Evol. Comput. IEEE Trans. 6 (2) (2002) 182–197, doi:10.1109/4235.996017.
[40] A. Arcuri, G. Fraser, On parameter tuning in search based software engineering, in: Proceedings of the 3rd International Conference on Search Based Software Engineering (SSBSE), 2011.
[41] J. Martinez, T. Ziadi, J. Klein, Y. Traon, 10th European Conference Modelling Foundations and Applications, Springer, pp. 117–131. doi:10.1007/978-3-319-09195-2_8.

[14] Such as Meta Object Facility (MOF) http://www.omg.org/spec/MOF/.

[42] S. Fischer, L. Linsbauer, R.E. Lopez-Herrejon, A. Egyed, Enhancing clone-and-own with systematic reuse for developing software variants, in: IEEE 30th International Conference on Software Maintenance and Evolution, in: ICSME, IEEE Computer Society, Washington, DC, USA, 2014, pp. 391–400, doi:10.1109/ICSME.2014.61.

[43] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Castor Filho, F. Dantas, Evolving software product lines with aspects: an empirical study on design stability, in: International Conference on Software Engineering (ICSE), ACM, 2008, pp. 261–270, doi:10.1145/1368088.1368124.