Contents lists available at ScienceDirect

Information and Software Technology

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

Combining multiple granularity variability in a software product line approach for web engineering

Jose-Miguel Horcas [a,*], Alejandro Cortiñas [b], Lidia Fuentes [a], Miguel R. Luaces [b]

[a] *Universidad de Málaga, Andalucía Tech, Spain*
[b] *Universidade da Coruña, CITIC, Fac. Informática, Database Lab. Elviña, 15071 A Coruña, Spain*

ARTICLE INFO

Keywords:
Annotations
Composition
Feature models
SPL
Variability
Web engineering

ABSTRACT

Context: Web engineering involves managing a high diversity of artifacts implemented in different languages and with different levels of granularity. Technological companies usually implement variable artifacts of Software Product Lines (SPLs) using annotations, being reluctant to adopt hybrid, often complex, approaches combining composition and annotations despite their benefits.
Objective: This paper proposes a combined approach to support fine and coarse-grained variability for web artifacts. The proposal allows web developers to continue using annotations to handle fine-grained variability for those artifacts whose variability is very difficult to implement with a composition-based approach, but obtaining the advantages of the composition-based approach for the coarse-grained variable artifacts.
Methods: A combined approach based on feature modeling that integrates annotations into a generic composition-based approach. We propose the definition of compositional and annotative variation points with custom-defined semantics, which is resolved by a scaffolding-based derivation engine. The approach is evaluated on a real-world web-based SPL by applying a set of variability metrics, as well as discussing its quality criteria in comparison with annotations, compositional, and combined existing approaches.
Results: Our approach effectively handles both fine and coarse-grained variability. The mapping between the feature model and the web artifacts promotes the traceability of the features and the uniformity of the variation points regardless of the granularity of the web artifacts.
Conclusions: Using well-known techniques of SPLs from an architectural point of view, such as feature modeling, can improve the design and maintenance of variable web artifacts without the need of introducing complex approaches for implementing the underlying variability.

1. Introduction

Web engineering involves managing a high diversity of artifacts at a different level of abstraction (e.g., web pages, templates, style sheets, code, or databases). Development companies profit greatly from component reusability and user-specific customization. To deal with the automatic generation of web applications, the industry advocates for adopting software product lines (SPLs) approaches [1,2].

A crucial step in SPL is modeling and implementing the variability of the reusable artifacts [3]. The most popular technique to model variability is feature modeling, which expresses the variability in terms of common and optional features [4]. For implementing variability, annotation-based approaches [3] are widely used in practice. They are simple, flexible, and easy to adopt because they only require annotating a common base artifact with the feature's variability information. In contrast to annotations, most of SPLs studies suggest that artifacts should be developed in the form of composable features which highly improve modularization, separation of concerns, reusability, and maintenance [3,5,6]. Nevertheless, the industry is reluctant to adopt composition-based approaches (*e.g.,* feature-oriented programming (FOP) [7], aspect-oriented programming (AOP) [8], or delta-oriented programming (DOP) [9]) because they require investing a high effort to be adopted successfully. Only some classic composition-based approaches [3] such as extensible frameworks with plug-ins, or service-oriented architectures are being used in the web engineering domain to support the reuse of a large granularity (*e.g.,* black-box web components). However, web applications must handle great amounts of fine-grained variability across different types of source code (Python, JavaScript, Java), templates (HTML, Markdown), style sheet files (CSS, and its variants such as SCSS), data serialization languages (JSON, XML,

* Corresponding author.
E-mail addresses: horcas@lcc.uma.es (J.-M. Horcas), alejandro.cortinas@udc.es (A. Cortiñas), lff@lcc.uma.es (L. Fuentes), miguel.luaces@udc.es (M.R. Luaces).

https://doi.org/10.1016/j.infsof.2022.106910
Received 5 April 2021; Received in revised form 23 February 2022; Accepted 22 March 2022
Available online 5 April 2022

YAML), and other kinds of files (property files or script files). The fine-grained variability that is scattered among several artifacts can be easily implemented with annotations but makes the maintenance of the SPL artifacts a complex and a potential error-prone task [5]. Moreover, the traceability of features and the affected artifacts is not easy in annotation-based approaches, even more considering the high diversity of languages involved in a web application.

Then, the first research question that arises is *how can an SPL approach handle the high diversity and granular variability of the web engineering domain?* (**RQ1**). Several works try to combine annotation and composition-based approaches to get their complementary benefits [5,10,11] (Section 2). These works attempt to handle coarse-grained variability with annotations, introducing feature composition into annotation-based approaches with the definition of new implementation layers, resulting in complex approaches to be adopted by the industry. In a previous work [12], we proposed just the contrary: instead of extending annotations with composition mechanisms, we proposed to integrate annotations into a composition-based approach using the Common Variability Language (CVL) [13]. CVL acted as a composition-based approach and was extended to manage fine-grained variability thanks to the plethora of variation points provided by CVL. However, the lack of tool support of CVL as well as its legal patent-related issues [14] have made CVL fall in disuse nowadays, making it difficult for the industry the adoption of CVL-based approaches [15]. In this paper, we answer RQ1 by proposing a combined approach that effectively integrates annotations into a generic composition-based approach by using feature models, instead of CVL, to handle the coarse and fine-grained variability of web applications. We define a mapping between the feature models and web artifacts, promoting the traceability of the features and the uniformity of the variation points regardless of the granularity of the web artifacts (Section 3).

Migrating an SPL in a company from a pure annotation-based approach to a new compositional approach is a problem that hinders the adoption of the combined approach. Web development companies develop SPLs mostly using annotations (*e.g.,* preprocessors [16] or configuration parameters) or defining their custom tool-based solutions. Maintaining and evolving an SPL following these strategies is an arduous task, as the number of annotated files grows exponentially and the quality of the annotated source code decreases. The second question that arises at this point is *is it feasible to migrate an annotation-based towards a combined (more composable-based) implementation?* (**RQ2**). To answer RQ2, we apply our combined approach to a case study based on a web-based SPL that currently implements its variable artifacts with annotations to handle both fine and coarse-grained variability [17,18]. The goal is to keep the annotations in the web SPL to implement the fine-grained variability which is difficult to implement with a composition-based approach, while at the same time improving the modularity of the code and the traceability between features, variation points, components, and final source files (Section 4).

Once that our combined approach is applied, our two last research questions are *how does the resulting combined approach perform compared to the previous annotation-based implementation of the SPL?* (**RQ3**), and *how does the resulting combined approach perform compared to the existing approaches* **RQ4**. We answer to RQ3 by evaluating our approach through the use of a set of metrics for analyzing variability and its implementation which have been recently surveyed in [19,20]. To answer RQ4, we evaluate the quality criteria [3] of our approach in comparison with the most relevant combined solutions of the state-of-the-art (Section 5).

2. State-of-the-art and motivation

One of the main objectives of SPL approaches is to derive a product automatically from variable code based on the user's requirements. To implement the variability in an SPL, several implementation techniques exist [3]. They can be classified into composition-based, annotation-based, and combined (hybrid) approaches. In the following, we present the most relevant variability implementation techniques and discuss their limitations in general, and especially, in the context of web engineering. Fig. 1 summarizes the existing techniques and their relations based on the classification proposed.

2.1. Composition-based approaches

Composition-based approaches implement features in the form of composable units (*e.g.,* modules, containers, classes, files), ideally mapping one feature to one unit. During product derivation, a combination of selected units, based on the required features, are composed to form the final product. There is a large body of research work on composition-based approaches (see left-hand side of Fig. 1), from classical frameworks with extensible plug-ins, component-based and service-oriented architectures, or even traditional design patterns to model variability, to specific programming paradigms such as feature-oriented programming (FOP) [7], aspect-oriented programming (AOP) [8], or delta-oriented programming (DOP) [9]. A detailed description of all these composition-based approaches can be found in [3].

Limitations. Composition-based approaches are rarely adopted in practice. This is mainly because composition-based approaches are challenging and error-prone. For instance, FOP and AOP, two of the most studied composition approaches in research, require that the industry takes risks and puts high efforts to successfully adopt these technologies [5,21]. Their corresponding tools (*e.g.,* FeatureIDE, AHEAD tool, AspectJ) have to meet high requirements and are hard to integrate with existing development processes. In particular, in the web engineering domain, the multilanguage nature of web applications, involving handling multiple types of artifacts and a great amount of fine-grained variability, makes it extremely difficult to apply some advanced programming paradigms (*e.g.,* FOP, AOP, DOP). There have been some attempts in the past, such as [22] in 2005 that presents a tool that generates HTML from XLST templates; or [23] in 2009, with an extension for Visual Studio that generates a .NET/ASP web page from UML standard diagrams. However, web applications are nowadays built using a completely different set of technologies since the rise of JavaScript popularity at the beginning of the 2010's decade. A more recent work, [24], uses DOP to implement microservices in Java, but it does not handle the user interface (UI), which is the most complex part of a web application. There are other composition-based approaches that rely on version control systems (*e.g.,* Git, Mercurial) and build systems (*e.g.,* ant, maven), which are very well known in the industry, but they are normally used for their intended purpose (*i.e.,* to control the version history of the products, or to compile and build the products), not for modeling variability in an SPL [3].

2.2. Annotation-based approaches

Annotation-based approaches (see right-hand side of Fig. 1) mark a common artifact to identify which part of the artifact belongs to a certain feature. During product derivation, all parts of the artifacts (*e.g.,* pieces of code) that do not belong to the required features are removed or ignored to form the final product. Preprocessors (*i.e.,* `#ifdef` annotations) [16] are the most widely used technique for implementing variable code, but also the use of configuration parameters can control and alter the behavior of a program (run-time variability) [3]. Other advanced approaches like virtual separation of concerns are tool-based and consist of a combination of tracing information, visualization facilities, and source code views (*i.e.,* coloring), to separately display, edit, and manage the code belonging to individual features [21].

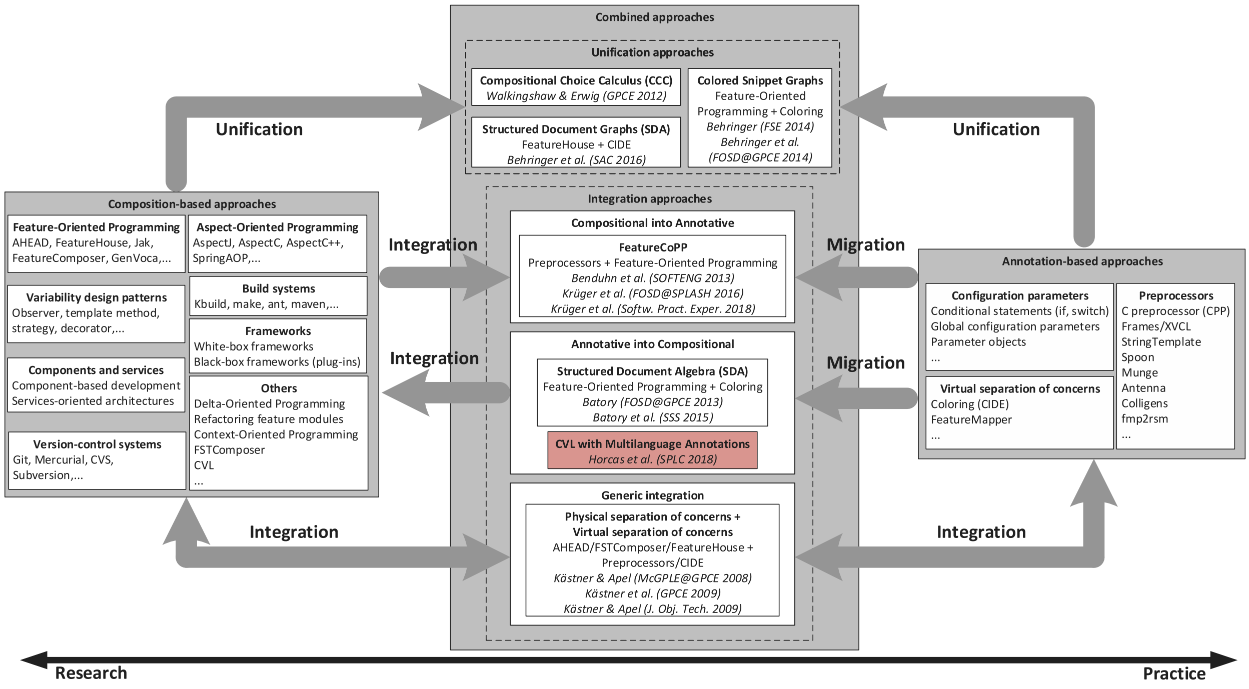

Combined approaches
Unification approaches
Compositional Choice Calculus (CCC)
Walkingshaw & Erwig (GPCE 2012)
Colored Snippet Graphs
Feature-Oriented Programming + Coloring
Behringer (FSE 2014)
Behringer et al. (FOSD@GPCE 2014)
Structured Document Graphs (SDA)
FeatureHouse + CIDE
Behringer et al. (SAC 2016)
Unification
Unification
Integration approaches
Compositional into Annotative
FeatureCoPP
Preprocessors + Feature-Oriented Programming
Benduhn et al. (SOFTENG 2013)
Krüger et al. (FOSD@SPLASH 2016)
Krüger et al. (Softw. Pract. Exper. 2018)
Annotative into Compositional
Structured Document Algebra (SDA)
Feature-Oriented Programming + Coloring
Batory (FOSD@GPCE 2013)
Batory et al. (SSS 2015)
CVL with Multilanguage Annotations
Horcas et al. (SPLC 2018)
Generic integration
Physical separation of concerns + Virtual separation of concerns
AHEAD/FSTComposer/FeatureHouse + Preprocessors/CIDE
Kästner & Apel (McGPLE@GPCE 2008)
Kästner et al. (GPCE 2009)
Kästner & Apel (J. Obj. Tech. 2009)
Composition-based approaches
Feature-Oriented Programming
AHEAD, FeatureHouse, Jak, FeatureComposer, GenVoca,...
Aspect-Oriented Programming
AspectJ, AspectC, AspectC++, SpringAOP,...
Build systems
Kbuild, make, ant, maven,...
Variability design patterns
Observer, template method, strategy, decorator,...
Frameworks
White-box frameworks
Black-box frameworks (plug-ins)
Components and services
Component-based development
Services-oriented architectures
Others
Delta-Oriented Programming
Refactoring feature modules
Context-Oriented Programming
FSTComposer
CVL
...
Version-control systems
Git, Mercurial, CVS, Subversion,...
Integration
Integration
Integration
Migration
Migration
Integration
Annotation-based approaches
Configuration parameters
Conditional statements (if, switch)
Global configuration parameters
Parameter objects
...
Preprocessors
C preprocessor (CPP)
Frames/XVCL
StringTemplate
Spoon
Munge
Antenna
Colligens
fmp2rsm
...
Virtual separation of concerns
Coloring (CIDE)
FeatureMapper
...
Research
Practice

Fig. 1. State-of-the-art of SPL variability implementation techniques: composition, annotative, and combined approaches.

Limitations. Annotation-based approaches are widely used in practice because they are simple, flexible, and are already natively supported by many programming languages. However, numerous studies criticize annotation-based approaches as they hinder the traceability and physical separation of features. Moreover, it is well known that the maintenance and evolution of the platform code using annotations is a nightmare because annotations obfuscate the base functionality of the application [3].

2.3. Combined approaches

To overcome the limitations of composition and annotative approaches, several hybrid approaches have emerged. Kästner and Apel [6] were the first who formulated the idea of combining both approaches. In principle, any combination of composition-based and annotation-based approaches is possible [21]. We classify the existing proposals into two groups: unification approaches and integration approaches (middle of Fig. 1).

2.3.1. Unification approaches

Unification approaches propose new paradigms to support both composition and annotations. They usually unify the concepts of two or more existing approaches into a new paradigm. Examples of unification approaches are the *compositional choice calculus* (CCC) [25], *colored snippet graphs* [26,27], and *structured document graphs* (SDA) [28]. Walkingshaw and Erwig [25] propose *compositional choice calculus* as a formal calculus model that unifies composition and annotations by generating editable documents (views) from a variability-aware *abstract syntax tree*. This approach has been put in practice [29] and depends on the programming language used. Behringer *et al.* [26,27] propose to unify composition (feature-oriented programming) and annotative (coloring) approaches with adapted tools: FeatureHouse [30] and CIDE [21]. In particular, they propose *structured document graphs* [28] based on the *compositional choice calculus* [25] to change between composition, annotations, and the combination of both approaches in an SPL.

2.3.2. Integration approaches

Integration approaches represent the concepts of a specific approach using another existing approach. For instance, representing compositional mechanisms using an existing annotation-based approach (*i.e.,* compositional into annotative), or representing annotations using an existing composition-based approach (*i.e.,* annotative into compositional). In the former, the goal is to exploit the benefits of the compositional mechanisms (*e.g.,* modularization, separation of concerns, reusability, and maintenance) in annotative approaches while maintaining the flexibility and easy adoption of the annotations. In the latter, the goal is to migrate from pure annotations towards a more compositional approach to achieve the major benefits provided by composition [5,6]. Almost all existing works that put into practice the combined approach in SPL are mainly based on the idea of the generic integrated (hybrid) approach proposed by Kästner and Apel [6].

- **Generic combination.** Kästner and Apel [6] analyzed and compared both composition and annotative approaches, separately in detail, and showed the benefits of an integrated approach, considering both compositional into annotative, and annotative into compositional approaches. They claim that the integration is straightforward, conceptually, and technically, since it is based on combining existing implementation techniques such as combining preprocessors [16] or virtual separation of concerns [31] with FOP [7], AOP [8], or DOP [9]. In particular, in [31,32] they introduce an additional implementation layer on top of preprocessors and CIDE to support compositional approaches like AHEAD, FSTComposer, and FeatureHouse.
- **Compositional into annotative approaches.** These approaches focus on implementing compositional mechanisms using annotations to overcome the problems of migrating from annotations to composition and refactoring an annotated SPL. For example, Benduhn *et al.* [11] apply the integration approach proposed by Kästner and Apel [6] in a real case study, by migrating Berkeley DB from C preprocessor annotations towards partial composition. They demonstrate that although the idea is feasible, the task is challenging, error-prone, and that not all physical separations can be achieved easily. Krüger *et al.* [5,10] present FeatureCoPP (Feature Compositional PreProcessor), an integrated implementation

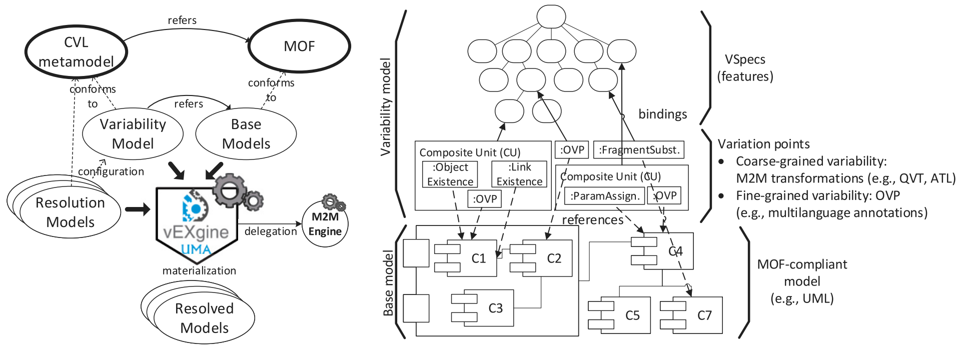


Fig. 2. The CVL approach integrating composition and multilanguage annotations.

concept that introduces composition into an annotation-based approach. Concretely, they extend the idea of preprocessors to support composition and enable physical separation of concerns similar to FOP, where the traceability is based on feature naming to identify features in different modules.

- **Annotative into compositional approaches.** These approaches integrate annotations into an existing compositional approach. For instance, Don Batory [33,34] proposes two algebraic models: the *feature interaction algebra* and the *structured document algebra* (SDA). These models formalize the concept of a module with variation points, the composition of them, and the decomposition of the modules into smaller parts, simulating annotations for FOP. In our previous work [12], we proposed to incorporate annotations into the Common Variability Language (CVL) [13] to allow the compositional approach of CVL to handle both coarse and fine-grained variability. The approach presented in this paper also fits in this category.

Limitations. The advantages of the combined approaches mainly depend on the specific composition and annotative implementation techniques used. However, the election of the programming model (*e.g.,* using or not using AspectJ for AOP) should not be imposed by the SPL implementation mechanism. Technological companies are reluctant to embrace those kinds of combined approaches, mainly when those approaches impose the adoption of a new programming paradigm or require the definition of new implementation layers, resulting in complex approaches to be adopted by practitioners. This is even worse if we need to apply a new programming paradigm in the context of web engineering because we need to use several languages and new languages appear on the market every day.

To overcome the limitations of the combined approach, in a previous work, we proposed a combined approach integrating CVL with multilanguage annotations [12]. The following section details the CVL approach along with its limitations. In this paper, we extend that work to make the approach independent from the CVL language (see Section 3).

2.4. The combined approach of CVL with multilanguage annotations

The CVL approach (Fig. 2) is, by nature, an orthogonal composition-based approach since artifacts can be composed, removed, or substituted through the CVL variation points. Variation points specify how the artifacts are modified by defining specific modifications to be applied using model-to-model (M2M) transformations to a base model. The semantics of these transformations are specific to the kind of each variation point. During CVL's execution, the CVL engine (vEXgine [35]) delegates its control to an M2M engine in charge of executing the transformations of each variation point. Only the semantics of those variation points bound to a selected feature in a configuration model will be executed during variability resolution.

The coarse-grained variability is managed by the variation points supported and predefined in CVL for composition. Some of these variation points are the existence of elements of the base model (*ObjectExistence*), the links between them (*LinkExistence*), the assignment of an attribute's value (*ParametricSlotAssignment*), or the replacement of a set of elements with another set of elements (*FragmentSubstitution*). The fine-grained variability is managed by defining a custom-made variation point (model transformation), that is, in CVL, a new Opaque Variation Point (OVP) which specifies the same semantics of the multilanguage annotations. Multilanguage annotations are a variant of `#ifdef` preprocessors [16] based on the technique of scaffolding [18], and allows marking any text-based artifact independently from the language used in such artifact (see Section 4.1).

Limitations. The main advantage of CVL to manage variability is that all types of artifacts are encoded and synthesized similarly as Meta-Object Facility (MOF) references, regardless of the implementation technique. This allows representing all artifacts subject to variation as software components of an architectural model (*e.g.,* in UML) and applying M2M transformations to resolve the variability. However, specifying all types of web artifacts in a MOF-based model is not always possible and will require defining a new MOF-compliant domain-specific language (DSL) for web engineering. Besides, custom M2M transformations in CVL need to be specified in QVT [13] or ATL [36] requiring practitioners to have a wide knowledge of model-based development (MDD) to define correct and generic transformations between MOF-compliant models. These are technologies that are not still widely adopted in industry [37]. The use of other transformation approaches requires extending the CVL execution engine as in vEXgine [35].

When modeling variability, the CVL variability model needs to be defined conform to the CVL metamodel [38], but the lack of tool support for modeling variability in CVL [35], along with its dreadful nomenclature of the modeling concepts (*e.g.,* VSpecs vs. features, resolution vs. configuration, materialization vs. product derivation, and the large set of variation points) makes the use of CVL an arduous task in practice. Moreover, its legal, patent-related issues [14] have made CVL fall in disuse nowadays, and the SPL community is betting high again for feature models as the de-facto standard for modeling variability [4].

A complete comparison based on different quality criteria between the most relevant combined approaches, including the CVL approach, and our approach is presented in Section 5.2.

3. A combined approach to model multiple granular variability

To answer RQ1: *how can an SPL approach handle the high diversity and granular variability of the web engineering domain?*, we need to manage fine-grained and coarse-grained variability of web engineering in an integrated approach. In a previous work, we successfully managed the variability with CVL and multilanguage annotations. In this section, we propose a new approach that replaces entirely CVL to promote the

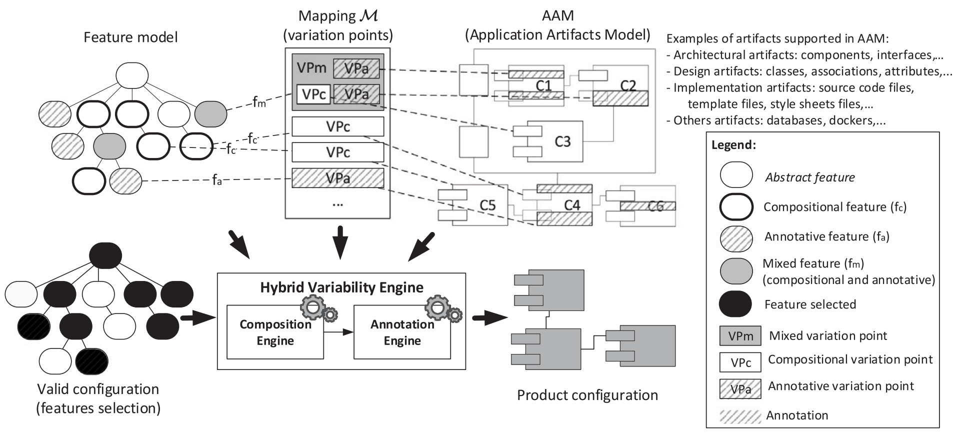

Feature model
Mapping $\mathcal{M}$ (variation points)
AAM (Application Artifacts Model)
Examples of artifacts supported in AAM:
- Architectural artifacts: components, interfaces,...
- Design artifacts: classes, associations, attributes,...
- Implementation artifacts: source code files, template files, style sheets files,...
- Others artifacts: databases, dockers,...
VPm
VPa
VPc
fm
fc
fa
C1
C2
C3
C4
C5
C6
Hybrid Variability Engine
Composition Engine
Annotation Engine
Valid configuration (features selection)
Product configuration
Legend:
Abstract feature
Compositional feature (fc)
Annotative feature (fa)
Mixed feature (fm) (compositional and annotative)
Feature selected
Mixed variation point
Compositional variation point
Annotative variation point
Annotation

Fig. 3. A combined approach to model multiple granular variability.

adoption of our approach. Our approach fits into the "annotative into compositional" integration approaches of the taxonomy presented in the previous section (Fig. 1).

3.1. Orthogonal variability modeling

Fig. 3 shows our approach where we specify separately, the application artifacts and the variability that can be applied to those artifacts. We denote the set of all application artifacts in the SPL domain as the *application artifacts model* (AAM).

Definition 1 (*Application Artifact Model*)**.** An application artifact model (AAM) is a software architectural model specifying the different artifacts of an application. That is, $AAM = \{a_1, a_2, a_3, \dots\}$ where a_i can be an arbitrary artifact representing, as for example, a component in an architectural model, a class in a class diagram model, a source code file, a database, or a web resource.

The AAM is equivalent to the *base model* used in the CVL language for MOF-compliant models like UML, however, in contrast to the CVL base model, AAM supports artifacts of different degrees of abstraction, from architectural models specified in UML to a source code file in Java. In the web engineering domain, the AAM can contain any kind of web artifact (constant or variable) that is required to build the web applications of the SPL. For instance, the AAM includes source code files like Java classes, JavaScript and Python code, HTML and markdown pages, style sheet files like CSS and SCSS, and other files such as JSON, YML, and shell scripts. Constant artifacts are common to all applications and do not contain any variability. Variable artifacts contain variability that can be implemented using composition or using annotations, based on their variability granularity, as we will explain in Section 3.3. The variability is specified at the abstract level separately in a feature model m [39]. The feature model specifies the set of features $F = \{f_1, f_2, f_3, \dots\}$ of the SPL, whether a feature is optional or mandatory, and the relationships between the features. A configuration c_m of the feature model is a valid selection of features that respects all the relationships and constraints specified in the feature model m. The configuration describes the features that compose a specific web application of the SPL.

3.2. Feature traceability and variation points

To link the feature model with the AAM, we define a mapping M as follows:

Definition 2 (*Mapping M*)**.** The mapping $M : F \rightarrow \mathcal{P}(AAM)$ is a function that maps each feature in F to the associated artifacts in the AAM implementing that feature. $\mathcal{P}(AAM)$ is the powerset of AAM, so that a feature in $f \in F$ can be mapped to an arbitrary number of artifacts $a \in AAM$, including the empty set in case that the feature has not associated artifacts (*e.g.,* an abstract feature or non-variable feature). The associated artifacts are variable and are subject to be modified when the variability is resolved to derive the final application. That is, the mapping M defines the variation points of the artifacts separately from the artifact's implementation.

Fig. 4 shows the Ecore-based metamodel with the abstract syntax definition of the mapping M and the variation points (VPs). Each variation point is bound to a feature f in the feature model and links to one or more artifacts a through a handle reference. Apart from the mapping information, each variation point defines a set of actions to be performed over the artifact to resolve its variability. The actions define the semantics (operations) of the variation points. Actions can be either model transformations to be executed by a model transformation engine (*e.g.,* Henshin [40], ATL [36], ETL [41]), or code to be executed by a programming execution engine (*e.g.,* a JavaScript engine, a C preprocessor engine). Finally, a variation point also contains a *negative variability* flag to indicate whether or not the operations of the variation point must be executed based on the selection or not of the bound feature in a configuration of the feature model. When the negative variability flag is set to *true*, the operations of the variation point will be executed if the bound feature is not selected in a configuration. This allows defining a negative variability approach for composition [35,42] in the same way that annotation-based approaches work with `#ifdef` preprocessor directives [16]. That is, starting from a complete application model we can remove elements from the model in case the associated feature is not selected in a configuration.

3.3. Managing fine and coarse-grained variability

The semantics (operations) of the variation points will depend on the variability granularity of the application artifacts. In our approach, we distinguish four kinds of artifacts based on their variability granularity: (1) common (non-variable) artifacts; (2) fine-grained variable artifacts; (3) coarse-grained variable artifacts; and (4) fine and coarse-grained variable artifacts. Common (non-variable) artifacts are those artifacts that do not present any variability. These common artifacts are present in all products of the SPL. Fine-grained variable artifacts are those artifacts that present variability in its internal structure, that is, they contain small pieces of code (like a code block, a few lines of sentences, or even a simple assignment statement) that are variable and can be customized in the SPL. Fine-grained variability is implemented as annotations [3]. Coarse-grained variable artifacts are those artifacts that are variable as a whole, that is, the whole artifact (a class, a complete file, or any modularity unit) can be present or not in a product of the SPL. Coarse-grained variability is implemented following

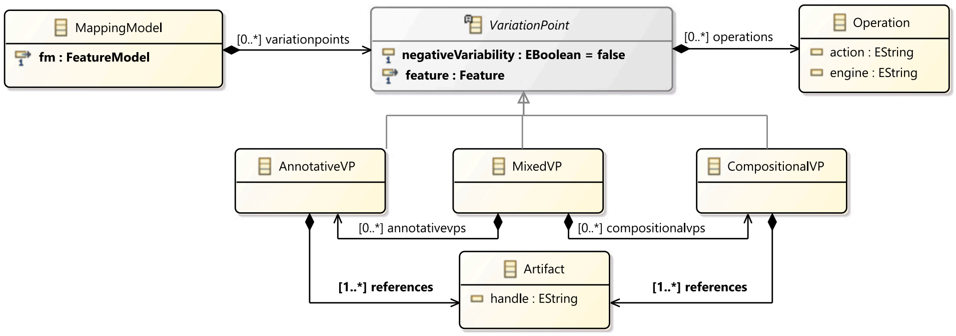


Fig. 4. Metamodel of variation points.

a compositional approach like FOP or AOP, but it may be implemented with annotations surrounding the whole artifacts [3]. Finally, fine and coarse-grained variable artifacts are those artifacts that are variable as a whole unit (coarse-grained variability), but also include variability in its internal structure (fine-grained variability). All these types of artifacts are illustrated in our case study in Section 4. Note also that in the literature there exists the concept of medium-grained variability [21], which refers to a variable function, or a method inside a class, or a well-identified code block. Medium-grained variability can be implemented as annotations (*e.g.,* an annotation surrounding a method) or following a compositional technique (*e.g.,* an aspect of AOP affecting a method) [3]. In our approach, medium-grained variability is considered as fine-grained variability, and artifacts containing variable functions/methods or variable code blocks are defined as fine-grained variable artifacts which are implemented with an annotation surrounding the affected code unit. To handle the different levels of granularity in our orthogonal approach, we define three types of variation points: compositional, annotative, and mixed (see metamodel in Fig. 4).

Compositional variation point (VP_c). Compositional variation points define the coarse-grained variability that is applied at the highest abstract level (*e.g.,* at the architectural level). Coarse-grained variable artifacts in the AAM model are composed, removed, substituted, woven, etc. to derive the final product. These are equivalent to traditional variation points provided by CVL such as *ObjectExistence*, *LinkExistence*, *FragmentSubstitution*, or *ParametricsSlotAssignment*. A difference with CVL (that only supports model transformations defined over MOF-based models) is that the operations of our compositional variation points can be defined as model transformations (*e.g.,* an ATL transformation rule), as operations over physical artifacts (*e.g.,* file system operations), or as specific composition operations used in composition-based approaches (*e.g.,* aspect weaving in AOP). The operation of the compositional variation points is executed by the composition engine associated with the variation points.

Annotative variation point (VP_a). Annotative variation points define the fine- and medium-grained variability that is applied at a lower level of abstraction (*i.e.,* at the artifact level such as source code or web templates). The semantics of a VP_a specifies that (1) there is an annotation bound to the selected feature; (2) the annotation (*e.g.,* a `#ifdef` statement, or a multilanguage annotation) is located in the component/artifact this variation point refers to; and (3) the annotation will be resolved by the annotation engine specified in the variation point (*e.g.,* our scaffolding-based derivation engine [18], *spl-js-engine*).

Mixed variation point (VP_m). Mixed variation points act as a container of variation points so that the feature bound to it is scattered across more than one variable artifact. Mixed variation points encapsulate both compositional and annotative variation points, each of them with its semantics and linked to the variable target artifact. When the hybrid variability engine (Fig. 3) finds a VP_m, it first resolves the variability of the compositional variation points (*i.e.,* coarse-grained variability), and then the annotative variation points of the remaining artifacts (*i.e.,* fine- and medium-grained variability).

Table 1 shows some examples of custom variation points defined using different operations and engines. For each variation point, we show the type (annotative or compositional), the negative variability flag, its description, an example of its operation, and the engine in charge of executing the operations. Note that our approach allows defining custom operations for variation points, so that the SPL developer can define and implement its own operations using any derivation engine. Our custom variation points can be similar to the Opaque Variation Points (OVPs) proposed in CVL, but OVPs only support operations specified as M2M transformations (*e.g.,* in ATL or QVT) [13,35], and thus, OVPs require an M2M engine to execute the operations. In contrast, as shown in Table 1, our approach allows specifying different operations for the same variation point. For instance, the `Existence` compositional variation point can be defined as a file system operation to be executed in a command-line interface (CLI) shell, or using an M2M transformation rule to be executed by an ATL engine [35]. Note also that Table 1 does not show any example of VP_m since it is just a container of VP_c and VP_a.

3.4. Product derivation

The center of Fig. 3 shows the hybrid variability engine, which is in charge of resolving the variability specified in a feature model over an AAM to derive a valid product configuration. The hybrid engine allows resolving both coarse- and fine-grained variability by delegating to the appropriate engine according to the information provided in each variation point. For convenience, we have implemented the hybrid engine as a CLI tool with *nodejs*, since it facilitates the invocation of the concrete engines.[1] For the composition engine we use shell scripting to resolve the coarse-grained variability represented in compositional variation points. While for the annotation engine we use the scaffolding-based derivation engine, *spl-js-engine* which was developed by the Databases Laboratory research group and it was used by its spin-off company, Enxenio in previous projects [17,18]. However, the hybrid engine also supports the delegation to other concrete engines, such as delegating to an ATL engine [35] for compositional variability using M2M transformation rules, or delegating to a C preprocessor [16] for annotative variability. The feature model, mapping model, application artifact model, and the valid configuration of a product are defined as JSON documents (for convenience, the feature model is also supported as an XML file following the schema of FeatureIDE [43]).

Algorithm 1 illustrates how the hybrid engine works to derive a final application product from the SPL. The hybrid engine will resolve first

[1] https://github.com/AlexCortinas/spl-js-engine

Table 1
Examples of custom variation points and their semantics.

VP	Type	Neg.	Description	Operations example	Engine
Existence	VP_a	No	It indicates the existence of a piece/block of code.	`/*% if (feature.<<name>>) { %*/ <<code block>> /*% } %*/`	Scaffolding
Existence	VP_a	Yes	It indicates the existence of a piece/block of code.	`/*% if (!feature.<<name>>) { %*/ <<code block>> /*% } %*/`	Scaffolding
Existence	VP_a	No	It indicates the existence of a piece/block of code.	`#ifdef <<feature.name>> <<code block>> #endif`	C preprocessor
Assignment	VP_a	No	It assigns a new value to an annotated variable/entity in the code.	`class /*= <<property.attribute>> */ { <<code block>> }`	Scaffolding
Existence	VP_c	Yes	It indicates the existence of an artifact.	`rm <<artifact.handle>>`	CLI (shell)
Existence	VP_c	Yes	It indicates the existence of an artifact.	`rule Existence { from a : AAM!Artifact (a.handle == <<this.handle>>) to drop }`	ATL (M2M)
Substitution	VP_c	No	It replaces an artifact with another one.	`cp <<source.handle>> . rm <<target.handle>> mv <<source.handle>> <<target.handle>>`	CLI (shell)

the coarse-grained variability (lines 3–6) and then the remaining fine-grained variability (lines 7–10). For each feature in the SPL, we obtain the associated variation points differentiating them into compositional (line 4) and annotative (line 8). That classification also considers variation points encapsulated in mixed variation points. The two separate loops ensure that all compositional variation points for all features are processed before considering any annotation variation points. The operations of the variation points are applied by default to those features selected in a valid configuration of the feature model (lines 13–19). However, this behavior can be modified with the *negative variability* flag for each variation point. This allows executing the operations of a variation point associated to a feature that is not selected, for example, to remove a complete artifact from the AAM when its bound feature has not been selected in the configuration. In Algorithm 1, the operations of a variation point will be executed only if the feature is selected in the provided configuration of the feature model in case of positive variability or if the feature is not selected in the configuration but the variation point has the negative variability flag activated (see line 15). To resolve the variability of a specific variation point (lines 20–27), we delegate the execution of its operations to the concrete engine specified in the variation point (line 23) (see Table 1). For instance, the operations of an `Existence` annotative variation point can be resolved using our scaffolding derivation engine (*spl-js-engine*) or a C preprocessor, according to the information provided in the variation point. That information includes the operation and the engine in charge of executing it, as well as the references (handles) to the artifacts, as specified in the metamodel of the variation points (Fig. 4). Note that before delegating the execution, we check for the validity of the artifacts' references specified in the variation point (line 22). As result, the hybrid engine returns a final application product, which is a concrete instantiation of the AAM model with all the variability resolved.

In the next section, we apply our combined approach to a running case study in the web engineering domain.

Algorithm 1 Product derivation process.

```
Input: feature model ($fm$), configuration ($conf$), mapping model ($map$), application artifact
    model ($aam$)
Output: application product ($app$)                    ▷ concrete instantiation of $aam$.
 1: function DERIVE_PRODUCT($fm$, $conf$, $map$, $aam$)
 2:     $app \leftarrow aam$
        ▷ First resolve compositional variability.
 3:     for each $f \in fm.features$ do
        ▷ Obtain compositional variation points bound to the feature (including $VP_c$ in $VP_m$).
 4:         $vps \leftarrow \{vp \in map.\text{variationpoints} \mid vp.\text{feature} = f \wedge vp \text{ instanceof } CompositionalVP\}$
 5:         $app \leftarrow$ PROCESS_VARIATIONPOINTS($f$, $vps$, $conf$, $app$)
 6:     end for

        ▷ Then resolve annotative variability for remaining artifacts.
 7:     for each $f \in fm.features$ do
        ▷ Obtain annotative variation points bound to the feature (including $VP_a$ in $VP_m$).
 8:         $vps \leftarrow \{vp \in map.\text{variationpoints} \mid vp.\text{feature} = f \wedge vp \text{ instanceof } AnnotativeVPs\}$
 9:         $app \leftarrow$ PROCESS_VARIATIONPOINTS($f$, $vps$, $conf$, $app$)
10:     end for
11:     return $app$
12: end function

13: function PROCESS_VARIATIONPOINTS($f$, $vps$, $conf$, $app$)
        ▷ Process the variability of the variation points associated with the feature.
14:     for each $vp \in vps$ do
        ▷ Check for positive/negative variability.
15:         if $(f \in conf \wedge \neg vp.\text{negativeVariability}) \vee (f \notin conf \wedge vp.\text{negativeVariability})$ then
16:             $app \leftarrow$ RESOLVE_VARIABILITY($vp$, $app$)
17:         end if
18:     end for
19: end function

20: function RESOLVE_VARIABILITY($vp$, $app$)
        ▷ Execute the operations associated with the variation point.
21:     for each $s \in vp.operations$ do
22:         if $\exists$ artifact $\in app \mid artifact.\text{handle} \in vp.\text{references}$ then
23:             EXECUTE($s$.action, $s$.engine, $vp$.references, $app$)    ▷ Delegate to concrete
    engine.
24:         end if
25:     end for
26:     return $app$
27: end function
```

4. Practical application

To test our proposal, we have designed and developed an SPL of web-based applications using an annotative-approach and a derivation engine we are familiar with, so we can afterwards apply the proposed approach to the same product line. The case of study chosen is an SPL for the generation of blogs, described in Section 4.1.

Then we describe how we manage to apply our approach, describing the different patterns of variation points and mapping that we have found integrating annotations into an orthogonal composition-based approach (see Section 4.2). Lastly, we briefly explain how the variability is resolved in order to derive a product using our proposed approach (see Section 4.3).

4.1. Case study: A blog SPL

Our case study is based on an SPL for the generation of blogs (see Fig. 5). A blog is a website where entries (called posts) are HTML text written by registered users of the blog using a post editor. The blog

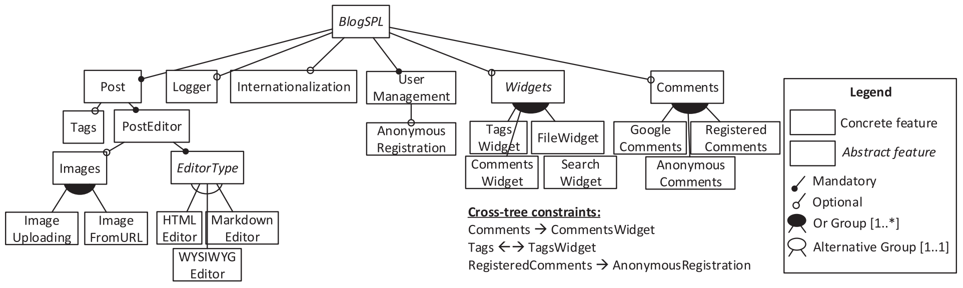


Fig. 5. Feature model of the blog SPL.

platform provides different types of editors to write the posts: an HTML, a Markdown, and a WYSIWYG editor. Posts can contain images that can be uploaded from a local file or straight from an external URL. Besides having an author and a timestamp, posts can also be linked with specific tags. To write a new post, a user needs to authenticate in the web application. The registered users are usually managed by an administrator. We can also allow anonymous users. Readers of the blog can comment on the posts, and we can even decide if they need to be registered users to comment or any anonymous user can do that. The blog can also have one or more widgets in the front page to manage the tags, comments, or files; and the user interface, including the administration pages, can be internationalized. Finally, as a means to debug the code properly, we can choose to add some extra logging in the code (*i.e.,* a logger). The features supported by this SPL are shown in Fig. 5, where they are organized using a feature model.

The *blog SPL* has been developed from scratch as a software engineering bachelor's thesis to help disseminate SPL concepts among students. The *blog SPL* was implemented following an annotative approach. The variability was resolved using the annotative-based derivation engine (*spl-js-engine*). The *blog SPL* was developed using modern web technologies, with a Spring[2] back-end exposing REST services, and a VueJS[3] front-end using them. Its base code is written in 8 different languages: JSON, Java, JavaScript, HTML, CSS, SQL, XML, and properties syntax.

Fig. 6 shows the software architecture of the blog SPL, including the variable artifacts with its granularity. In the current implementation, all the variability is directly implemented within the artifacts by using multilanguage annotations. There are artifacts containing annotations of many features. Such case appears, for example, in the `PostREST` component, or the `PostEditor` component. The former is composed by 8 files (Java classes and interfaces), containing 40 annotations of 10 features (`Post`, `HTMLEditor`, `MarkdownEditor`, `WYSIWYGEditor`, `Tags`, `Images`, `Comments`, `FileWidget`, `TagsWidget`, and `SearchWidget`). `PostEditor` component is composed itself by 3 files (VueJS components, which includes HTML, JavaScript and CSS), and it contains 35 annotations of 7 different features (`PostEditor`, `Comments`, `HTMLEditor`, `MarkdownEditor`, `WYSIWYGEditor`, `Tags`, and `Images`). There are other components which are the opposite, they have only a feature related with little annotations. This is the case for the componentes `Tags Widget`, `AnonymousCommentsEditor` or `Tag Addon`, for example. The two first are each one composed by 1 file (a VueJS component), with 1 annotation of 1 feature (`TagsWidget` and `AnonymousComments`, respectively), while the latter is composed by 2 files (VueJS components), each one with 1 annotation of the feature `Tags`. Moreover, most of the features are scattered across many components, such as the feature `ImageUploading`, with annotations in the `ImageUpload-Service` in the server side, and the `ImageUploadClient` and `ImageModalEditor` in the client side. In some cases, the annotations handle coarse-grained variability (*e.g.,* by marking the whole file) like in the `FileWidget` artifact; in other cases, annotations handle fine-grained variability (*e.g.,* by marking a few lines of code as in Fig. 7) like in `Router` artifact; and finally, there are also artifacts that contain annotations handling both fine and coarse-grained variability like the `CommentViewer` or `CommentEditor` artifacts. It is important to note that this classification by granularity level is done after the SPL was developed with the purpose of better understanding the software product line architecture (SPLA), but in fact the annotation engine does not classify in any way the annotated source files, and it treats any file the same way.

Multilanguage annotations. An example of multilanguage annotations is illustrated as part of our case study in Fig. 7. We show simplified excerpts of annotated code for the `ImageModalEditor` artifact, which functionality is associated with the inclusion of images in a post, (a) for the view (HTML), and (b) for the controller (JavaScript).

Note that multilanguage annotations are embedded within comments and usually do not interfere with the source code, making them perfect to annotate any type of web artifact (HTML, CSS, Java, etc.). For instance, for HTML code, the annotations are within HTML comments. In case of the artifact was specified in any other language, the annotations would be inside the specific language syntax for comments. This allows customizing the delimiters for the annotations depending on the file, by linking each file extension with a particular delimiter. Developers would prefer to use the syntax of the comments that are native in every language. This way, they can work with their favorite IDE and tools without having to deal with intrusive annotations that make the code not compile or that will be marked as syntax errors by for example an HTML editor. There are two exceptions: a) fine-grained annotations, such as annotations that add a new parameter to a method or similar, can break language-related analysis for some IDEs or compilers; and b) very specific frameworks that mix languages in the same source file and do not uniform the comment delimiters, like in the case of VueJS single file components.[4] Anyway, we have not found any annotation-based alternative supporting multilanguage annotations that properly solves these issues.

Despite the benefits of multilanguage annotations, the great number of annotations makes it difficult to maintain and evolve the blog SPL. The goal is that the blog SPL continues having their annotations to handle fine-grained variability, but improving the modularity of the code and the traceability between features, variation points, and web artifacts (final source files), by using a generic (implementation-independent) composition technique, as the presented in Section 3, to handle the coarse and middle-grained variability.

[2] Spring framework: https://spring.io/
[3] VueJS framework: https://vuejs.org/
[4] For this specific case, the problem can be avoided by not using the single file component syntax.

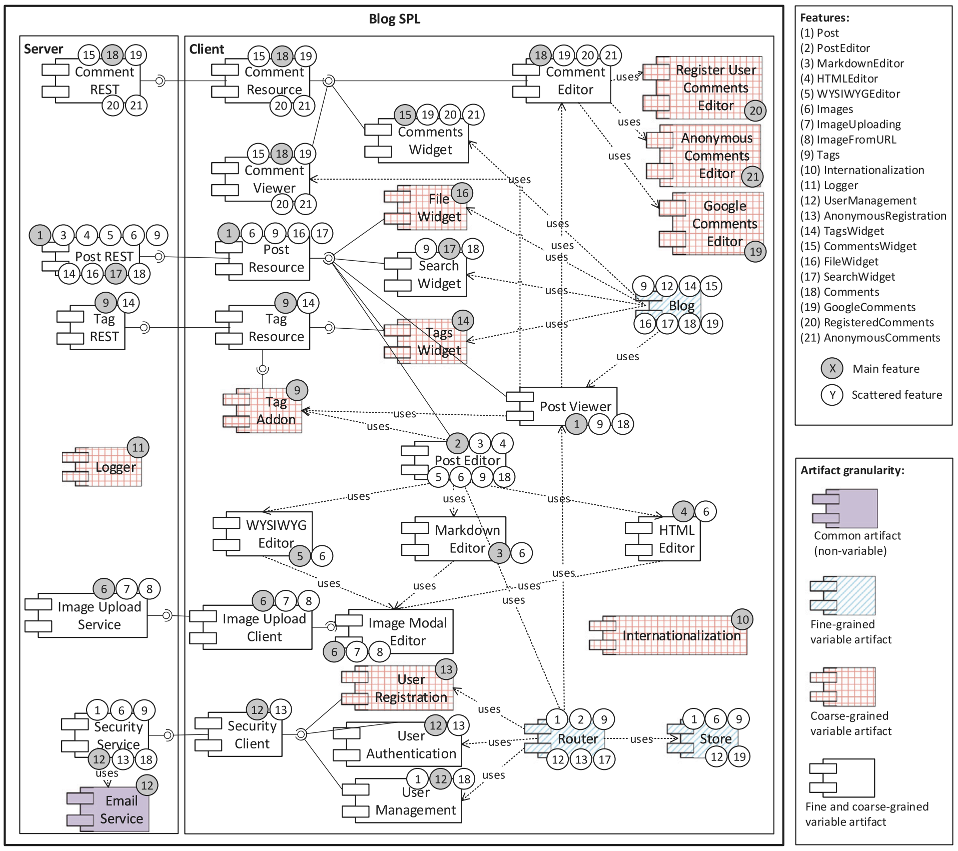


Fig. 6. Software architecture of the blog SPL.

4.2. *Integrating multilanguage annotations into an orthogonal composition-based approach*

This section answers RQ2: *is it feasible to migrate an annotation-based towards a combined (more composable-based) implementation?*. Our proposed approach, like CVL, is intentionally a compositional approach that is applied at a high level of abstraction like the architectural level instead of working at the code level. However our approach is unaware of how the features are physically separated into code units implementing the web artifacts (*e.g.,* components, aspects, feature modules), and therefore, any classical composition-based approach at the code level could be applied, such as for instance, FOP or AOP. Assuming features are separated the best possible in code units, an annotative approach can be used to additionally annotate code units when the variability affects finer levels [6]. So, one code unit can implement a variable feature and, at the same time, contain variable code text. The problem is that in web applications sometimes it is not possible to physically separate independent features in different code units, as desirable (see the current architecture in Fig. 6). Also, a web developer can be reluctant to adopt our new approach and prefers to continue using annotations to implement variable features at the code level. To handle all these cases we integrate annotations (our multilingual annotations) within the orthogonal composition-based approach presented in Section 3, to handle the fine-grained variability from a high abstract level.

Fig. 8 shows how our approach is applied to the blog SPL. We can distinguish three separate models: (1) the feature model specifying the variability of the blog SPL's features (top of Fig. 8), which is the same used in our annotation-based blog SPL (Fig. 5); (2) the mapping model M (middle of Fig. 8) specifying the information of the variation points, including our three types of variation points (compositional, annotative, and mixed); and (3) the AAM specifying the web artifacts that implement the blog SPL (bottom of Fig. 8). The three models are related through the variation points defined in the mapping M. Each variation point is bound to a unique feature in the feature model and references one or more artifacts in the AAM model. Note that to simplify the figure we only show an excerpt of the AAM, the complete blog SPL is evaluated in Section 5.

To build the mapping model M, we identify different application cases based on the granularity of the features and their current implementation across the web artifacts in the blog SPL (Table 2). For each case in Table 2, we expose the pattern that relates features with the artifacts, and the corresponding entry in our mapping model M that should be defined. The mapping entry contains the variation point (VP), the negative variability flag (NegV), its operation, and an example taken from the blog SPL architecture that matches the pattern. While cases C1–C3 refer to the granularity of a unique feature scattered across one or more artifacts from the feature point of view (`feature perspective`), cases C4–C7 refer to the implementation of two or more features in a unique artifact from the artifact point of view (`artifact perspective`). Complex cases can be addressed as the consecutive application of these patterns.

C1. Coarse-grained variable artifacts that only contain code belonging to one (compositional) feature and their variability is currently implemented by marking the entire artifacts with a multilanguage annotation. This is the case of the `Internationalization` and the `Logger` components (features 10 and 11 in Fig. 6 respectively). The annotations of those artifacts can be completely removed with the following steps: (1) creating a new compositional variation point (VP_c) in our mapping model M with the negative variability flag activated; (2) associating the feature (*e.g.,* `Logger`) of the feature model with the variation point; (3) referencing the artifact(s) — *e.g.,* `Logger` — with the variation point; and (4) assigning the operation `Existence`

```html
 <b-modal v-model="showModal" id="modal-center" size="lg" centered no-close-on-backdrop
     no-close-on-esc @ok="submit" @cancel="close" @close="close" @shown="getImagesGalleryInit"
     :cancel-title="$t('button.cancel')">
   <div slot="modal-title">{{$t("title.imageModal")}}</div>
   <div class="custom-modal">
    <b-tabs content-class="mt-3" no-key-nav ref="tabs">
      <!--% if (feature.ImageUploading) { %-->
      <b-tab ref="uploadTab" :title="$t('tab.uploadImage')" active class>
        <!-- implementation of the template for the feature imageUploading -->
      </b-tab>
      <!--% } %-->
      <!--% if (feature.ImageFromURL) { %-->
      <b-tab ref="urlTab" :title="$t('tab.urlImage')">
        <!-- implementation of the template for the feature imageFromURL -->
      </b-tab>
      <!--% } %-->
    </b-tabs>
  </div>
</b-modal>
```

(a) Simplified excerpt of the `ImageModalEditor` template (annotated HTML code).

```javascript
export default {
  name: "ImageModalEditor",
  /* omitted code */
  methods: {
    submit() {
      /*% if (feature.ImageUploading && feature.ImageFromURL) { %*/
      this.submit(this.$refs.tabs.currentTab.title);
      /*% else if (feature.ImageUploading) { %*/
      this.submitUpload();
      /*% else if (feature.ImageFromURL) { %*/
      this.submitUrl();
      /*% } %*/
      this.closeModal();
    },
    /*% if (feature.ImageUploading && feature.ImageFromURL) { %*/
    submit(type) {
      switch (type) {
        case this.$t('tab.uploadImage'):
          return this.submitUpload();
        case this.$t('tab.urlImage'):
          return this.submitUrl();
      }
    },
    /*% } %*/
    /*% if (feature.ImageUploading) { %*/
    submitUpload() { /* implementation of the feature ImageUploading */ },
    /*% } %*/
    /*% if (feature.ImageFromURL) { %*/
    submitUrl() { /* implementation of the feature ImageFromURL */ },
    /*% } %*/
    /* omitted code */
  }
};
```

(b) Simplified excerpt of the `ImageModalEditor` controller (annotated JS code).

Fig. 7. Example of multilingual annotations for two artifacts in different languages.

("remove artifacts") to the variation point, which mean that when the feature `Logger` is selected in a configuration, the final product will include all those referenced artifacts providing the functionality of the `Logger` feature, while if the feature `Logger` is not present in a configuration, all the referenced artifacts will be excluded from the final product.

C2. A feature is scattered across several artifacts, some of them having annotations handling coarse-grained variability and others with annotations handling fine-grained variability. An example of this pattern is the `Tags` feature which is implemented in the `TagREST`, `TagResource`, and `TagAddon` artifacts, and used or referenced in the `PostViewer`, `PostEditor`, and `SearchWidget` artifacts. So, `Tags` is not a pure compositional feature, but a mixed feature. In this case, we need to create a mixed variation point (VP_m) containing (1) a compositional variation point (VP_c) for those artifacts with coarse-grained annotations (as in case C1), and (2) an annotative variation point (VP_a) for those artifacts with fine-grained annotations.

C3. Several artifacts have annotations handling fine-grained variability of a feature. In our running case study, there are two examples of this kind: features `ImageUploading` and `ImageFromURL` modify the components `ImageUploadClient`,

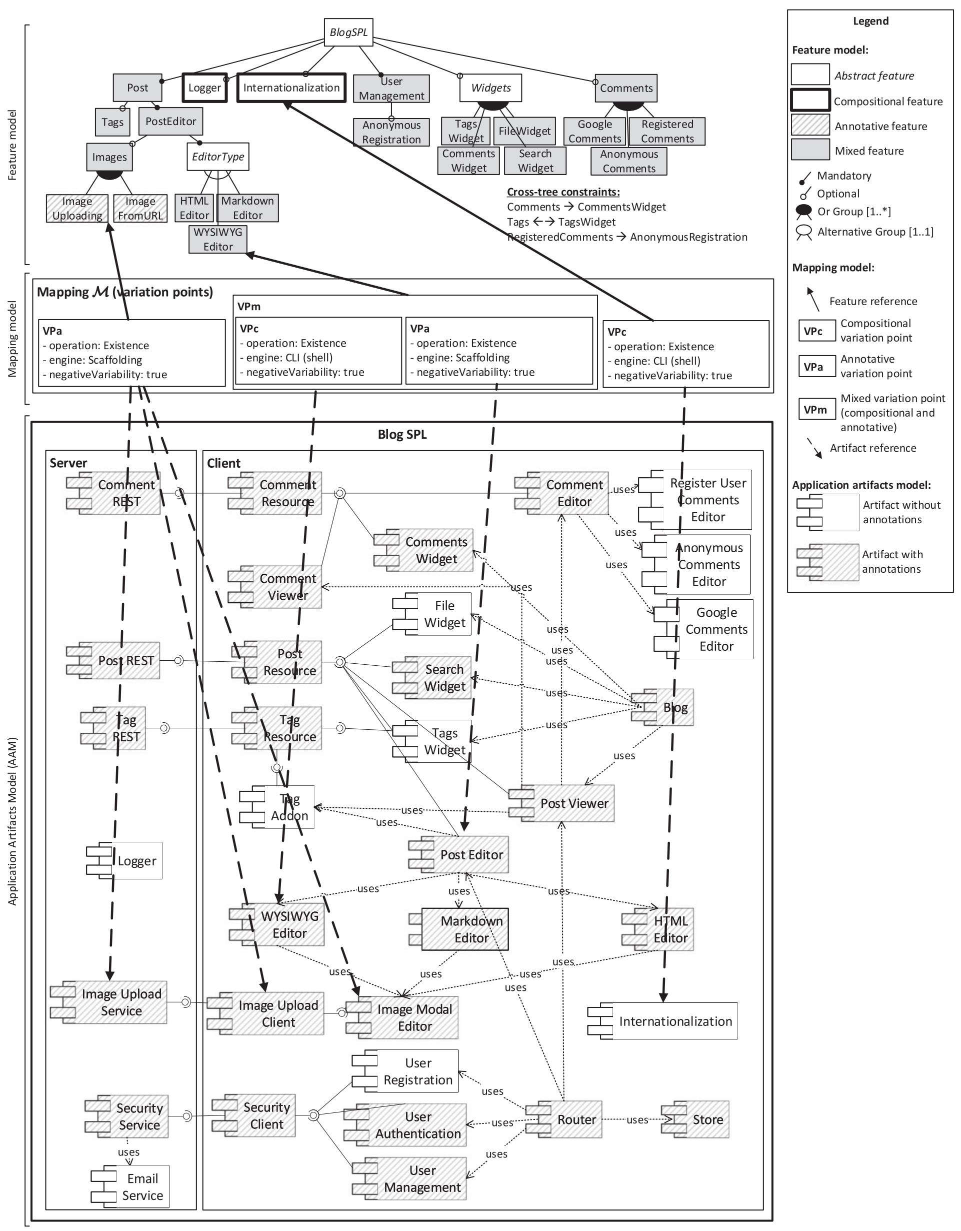


Fig. 8. Our approach applied to the blog software product line: feature model variability (top), application artifacts model (bottom), and mapping model (middle).

`ImageModalEditor` and `ImageUploadService`. These annotations remain unchanged in our approach at the artifact level, but we define a (VP_a) in the mapping model to trace the feature and the affected artifacts.

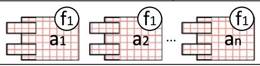
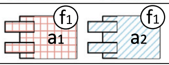
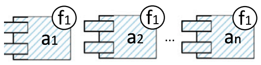
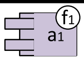
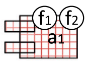
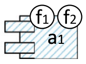
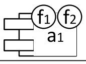
C4. Non-variable (common) artifacts do not need any entry in the mapping model because those artifacts will be present in all products of the SPL. An example of common artifact is the `EmailService` component, which is associated with the `UserManagement` mandatory feature.

C5. An artifact contains coarse-grained annotations of more than one feature. This case corresponds to a bad design in the implementation of the features and can be seen as a smell code that should be refactored. In our case study there is not an example of this case, but, our approach is unaware of the implementation details and can handle this kind of design by defining a VP_a for each feature. Thus, we maintain the annotations in those artifacts but improve the traceability of the involved features.

C6. An artifact contains fine-grained annotations of several features, as occurs in the `Router` artifact for features `Post`, `PostEditor`, `Tags`, `UserManagement`, `Anonymous Registration`, and `SearchWidget`, or in the `Store` artifact for features `Post`, `Images`, `Tags`, `UserManagement`, and `GoogleComments` (Fig. 6). For each different feature, we define a VP_a associated with the corresponding artifact, indicating that such artifact has annotated code which belongs to that feature.

C7. An artifact contains both coarse and fine-grained annotations referring to different features. This is a generalization of the previous cases, and specifically, a combination of cases C5 and C6, where some mixed features (compositional and annotative) are easier implemented using annotations but which also requires the inclusion of a component. For instance, artifacts `CommentViewer`, `CommentResource`, or `CommentREST` contain a coarse-grained annotation belonging to the `Comments` feature (number 18), while the same artifacts also contain fine-grained annotations belonging to the `GoogleComments`,

Table 2
Integrating annotations into an orthogonal composition-based approach.

	Case	Features and artifact implementation	Description	Mapping	VP	NegV	Semantics	Example (Features → Artifacts)
Feature perspective	C1		A coarse-grained feature implemented in one or multiple artifacts.	$\mathcal{M}$: f1 → {a1,a2,...,an}	VPc	true	"remove artifacts"	Logger → Logger
	C2		A mixed feature scattered across multiple artifacts.	$\mathcal{M}$: f1 → {a1,a2}	VPm [VPc VPa	true true	"remove artifacts" "remove annotated code"	Tags → TagAddon Tags → PostViewer
	C3		A fine-grained feature affecting one or multiple artifacts.	$\mathcal{M}$: f1 → {a1,a2,...,an}	VPa	true	"remove annotated code"	ImageFromURL → ImageUploadClient, ImageModalEditor, ImageUploadService
Artifact perspective	C4		Non-variable artifact.	Not needed	-	-	-	UserManagement → EmailService
	C5		Two or more coarse-grained features tangled in an artifact. Smell code. It is better to refactor.	$\mathcal{M}$: f1 → {a1} $\mathcal{M}$: f2 → {a1}	VPa VPa	true true	"remove annotated code" "remove annotated code"	- -
	C6		Two or more fine-grained features tangled in an artifact.	$\mathcal{M}$: f1 → {a1} $\mathcal{M}$: f2 → {a1}	VPa VPa	true true	"remove annotated code" "remove annotated code"	PostEditor → Router Post → Store
	C7		Two features with different granularity affecting one artifact.	$\mathcal{M}$: f1 → {a1} $\mathcal{M}$: f2 → {a1}	VPc VPa	true true	"remove artifact" "remove annotated code"	Comments → CommentViewer GoogleComments → CommentViewer

RegisteredComments or AnonymousComments features (features 19, 20 and 21 respectively). As in the previous cases, we define a VP_c for each feature implemented as coarse-grained annotations (removing the annotation from the artifact) and a VP_a for each feature implemented as fine-grained annotations (maintaining the annotation in the artifact).

The artifacts of the resulting AAM in Fig. 8 with regard to the original architecture presented in Fig. 6 have changed as follows. Common artifacts remain the same since they did not contain any variability. Coarse-grained variable artifacts are now artifacts without annotations because their variability has been defined compositively in a VP_c. Fine-grained variable artifacts and mixed (fine and coarse-grained) variable artifacts still contain annotations referring to the fine-grained variability, despite defining annotative variation points (VP_a) and mixed variation points (VP_m) for those artifacts respectively. Note that since the artifacts of the blog SPL already contain all the complete functionality of the SPL, we follow a negative variability approach where we remove specific functionality from a core complete implementation. Thus, all our variation points have the negative variability flag activated and use the same operation of the Existence variation point (*i.e.,* removing functionality) as specified in Table 1. Other variation points with different operation (*e.g.,* weave, add, or replace a new component) can be defined similarly.

From the point of view of the features, our mapping model looks as in Fig. 8 where we provide complete traceability for features. For instance, feature WYSIWYGEditor can be implemented with barely a few lines added to the JavaScript source code of the PostEditor component. However, it also requires a specific component able to show a preview of the user's text (WYSIWYGEditor component). All the variability of the WYSIWYGEditor feature is encapsulated in a mixed variation point (VP_m), and thus, when this feature is selected in a configuration (or not selected for negative variability), all variation points will be applied together. Note that the order in which the variability is resolved does not affect the final product, but it impacts the performance of the derivation process. Our hybrid engine first resolves the compositional variation points, and then the annotative variation points. This way we prevent resolving fine-grained variability of components that will be not present in the final product. An advantage over the CVL approach is that we can reuse a variation point over multiple artifacts if the variation point shares the same operations for those artifacts (see the ImageUploading feature and the associated VP_a in Fig. 8). In contrast, in CVL we need to define a variation point for each artifact, despite those artifacts belong to the same feature.

4.3. Resolving the variability to derive a product

In order to resolve the variability and derive a final product from the Blog SPL using our approach (Fig. 3), a specific configuration of the feature model must be provided as input of our Hybrid Variability Engine. A valid configuration of the feature model is depicted in Fig. 9 where the selected features have been highlighted. The engine also receives the feature model, the AAM, and the mapping model as specified in Fig. 8.

Following the Algorithm 1 the engine traverses all features in the feature models obtaining the associated variation points. The operations of the variation points will be executed if the feature is selected in the configuration (and the negative variability flag is deactivated) or if the feature is not selected in the configuration but the negative variability flag is activated. For example, in the configuration presented in Fig. 9 the Internationalization feature has not been selected, and its associated variation point (a VP_c with an Existence operation) has the negative variability flag activated (see Fig. 8). Therefore, the Internationalization artifact will be removed from the final product. The engine will execute first the operations of the compositional variation points, and then the operations of the annotative variation points. Annotative variation points inside mixed variation points will be executed only if the referenced artifacts exist after executing the compositional variation points inside the mixed one. For instance, the HTML Editor feature, which has not been selected, has bound a VP_m similar to the VP_m associated with the WYSIWYG Editor feature (which is selected in the configuration). The referenced artifact HTML Editor will be removed from the final product, but also the pieces of code related to that feature that are presented in the PostEditor artifact, since the PostEditor will be present in the final product. Fig. 10 shows a final Blog product according to the configuration provided in Fig. 9. Artifacts with coarse-grained variability such as the Internationalization component has been removed, while artifacts with fine-grained variability such as the PostEditor component has been configured by removing the unnecessary (annotated) pieces of code.

5. Evaluation

These sections evaluate our approach by answering RQ3: *how does the resulting combined approach perform compared to the previous annotation-based implementation of the SPL?* (Section 5.1), and RQ4: *how does the resulting combined approach perform compared to the existing approaches?* (Section 5.2). To answer these RQs we first apply our approach to the complete blog SPL and quantitatively evaluate it

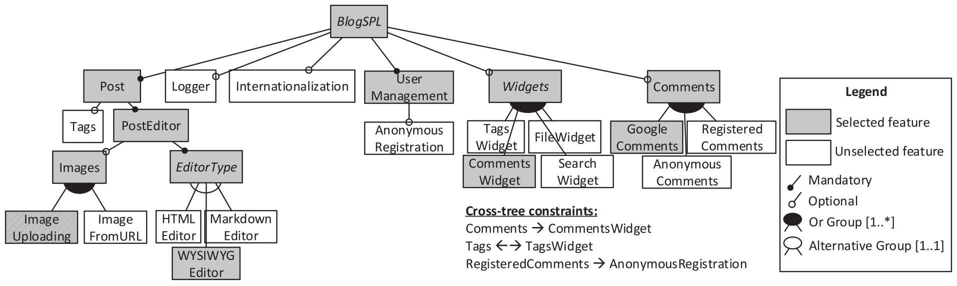


Fig. 9. A valid configuration of the Blog SPL feature model.

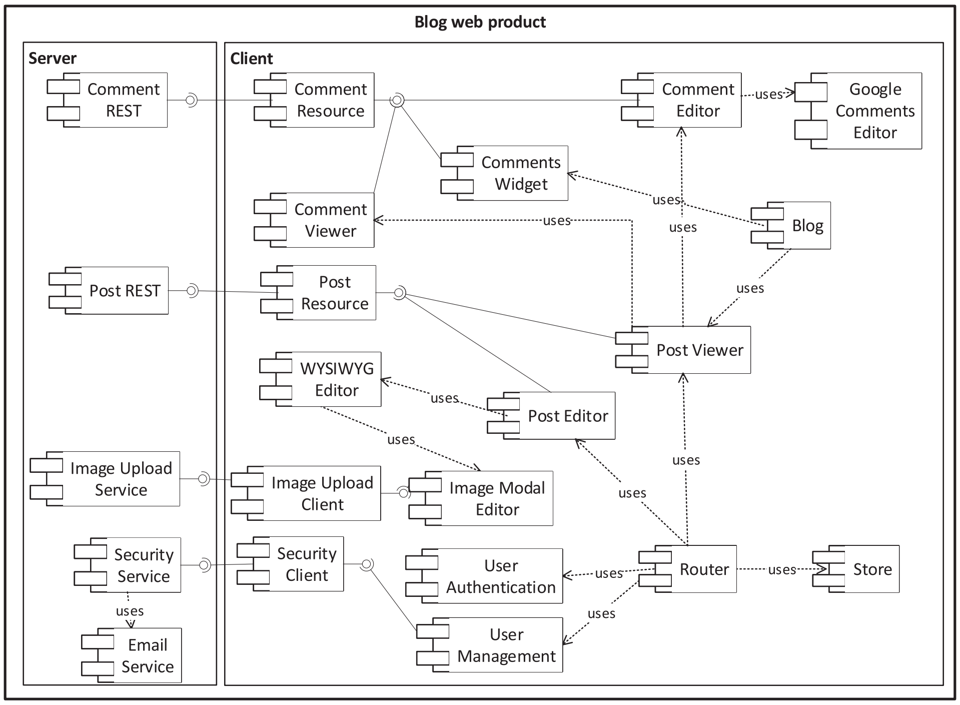


Fig. 10. Final blog product with the variability resolved.

using a set of metrics for variability and SPL [19,20]. Then, we also qualitatively evaluate our approach by discussing different quality criteria for SPL in comparison with the existing SPL implementation strategies presented in Section 2.

5.1. Quantitative evaluation

We compare our approach with the previous solution based only on annotations. A quantitative comparison with the previous combined approach using CVL [12] is direct, since its benefits over the pure annotated-approach are the same as in the combined approach presented in this paper. Note that one of the goals was to replace CVL with feature models due to the CVL limitations discussed in Section 2.4. Nevertheless, we qualitatively compare our approach with the CVL approach by discussing the quality criteria being affected.

[19,20] review an exhaustive collection of metrics for analyzing variability and its implementation from the literature. From this collection, we have selected those metrics that make sense to be applied in our context, a web-based blog SPL with around 17 K lines of code (LOC) and 35 components, including both server-side and client-side artifacts, and we have omitted the metrics that do not offer any interesting comparison. Particularly, the main reasons to exclude metrics from our evaluation were: (1) they are language-related, this is, their calculation is based on elements that exist on a specific object-oriented language, usually Java, and that do not make sense in other languages such as HTML or JavaScript (e.g., *aspect size*, *lack of cohesion in methods* or *class fragmentation*); (2) they do not make sense with both our approaches, this is, we are trying to compare an annotation-based approach with our combined approach so the metric should work for both (e.g., *number of refinements*); and (3) they are not affected by the new approach, this is, the result of the metric is the same independently of the approach (e.g., *cross tree-constraints ratio* or *number of valid configurations*).

The selected metrics are shown in Table 3, being most of them from the groups *composition* and *annotation-based code* [19], since metrics from the other two groups, *variability model* and *mixed*, were omitted for the given reasons. The metrics mostly show the difference between the approaches related to the percentage of the code which is in fact annotated, being counting lines of code (LOC), files or components. It can be observed that the lines of annotated code (LOAC) represent the 90,2% of the total in the annotation-based approach. This number goes down to the 16,7% of the total in our combined approach. In fact, the LOAC are reduced to the 18,38%, from 15 869 to 2 916, when switching to our new approach. The number of variable files, this is, files which includes at least one annotation, is also reduced but to a lesser extent (from 100 files to 62 files, over 117 total files for both approaches, since the actual source code does not change). Lastly, applying the same metrics with a different granularity, components instead of files, we see that the results are that more discrete since most of the components

Table 3
Metrics for analyzing variability and its implementation in SPLs.

Metrics	Annotation-based approach	Our combined approach
Number of files for the complete system	117	117
Number of features	24	24
Number of components	35	35
Lines of code (LOC)	17 576	17 400
Lines of annotated code (LOAC)	15 869	2 916
Number of distinct feature constants, this is, distinct features used in annotations, or concrete features	21	20
Number of variation points (#annotations)	450	362
Average number of annotations for a feature constant	21,43	18,10
Number of variable files for the complete system, or files with any annotation	100	67
Average of variable files belonging to a single feature	4,17	2,79
Number of variable components for the complete system, or components with any annotation	35	25
Average of variable components belonging to a single feature	1,46	1,04
Ratio of variable files over all files	85,47%	57,26%
Ratio of variable code lines over all lines of code	90,20%	16,70%
Ratio of variable components over all components	100%	71,42%

still include some variability. There is only a reduction in variable components of a 28,58%, corresponding mostly to components that include only one or two constant files with the new approach.

5.2. Qualitative evaluation

This section discusses and compares our approach to the pure composition and annotation-based approaches and to the most relevant combined approaches [6,10,12] presented in Section 2 when dealing with variability in the web engineering domain. We use the quality criteria for SPL implementation techniques defined in [3]: *feature traceability*, *separation of concerns*, *information hiding*, *granularity*, *uniformity*, and *preplanning effort*. We also incorporate others interesting quality criteria that are recommendable for SPL implementation, such as the support for multiple *languages*, the *variability type* supported, *automation*, *maintainability*, *evolution*, and *tool support*. The results are summarized in Table 4 with approximated grades following a Likert scale [44] with five points that express the level of support of each approach with the quality criteria. A value of 1 (↓) means that the approach does not provide enough support; and a value of 5 (↑) indicates that the approach provides very good support. We also show the main characteristic of the approach for each quality criteria according to the score assigned.

Feature traceability. Feature traceability describes the mapping between a feature in the variability model and its implementation in an artifact or set of artifacts. Approaches should offer the ability to explicitly locate features in software artifacts. The mapping helps developers to identify relevant artifacts during development and maintenance, and such mapping can be realized with several mechanisms (↑: binding and references, ↖: naming conventions, ←: naming conventions and tool support, ↙: depend on tool support, ↓: no traceability). For compositional approaches, this mapping depends on the implementation technique. It is said [3,10] that the mapping is direct as the artifact that implements a feature can be traced to a single code unit (component, module, aspect,…). However, this 'direct' trace is implicitly done by name conventions since the only way to identify and relate the feature in the variability model and the artifact that implements that feature is using the same identifiers for the feature and the artifact, and/or using dedicated tools [21,46]. Moreover, in annotative approaches, feature traceability is poorly supported because annotations can be scattered over multiple artifacts, and traceability, in this case, is usually a matter of tool support [3]. Traceability in combined approaches is weaker than in pure compositional approaches because existing integrating approaches [6,10] are annotative approaches that try to introduce composition. *Our approach defines a mapping model (see Fig. 3) that provides explicit traceability between the features and the application artifacts by defining variation points (as in CVL) that bind each feature and reference the artifacts implementing that feature. Although an annotative feature is scattered in multiple artifacts due to a bad design, our approach allows explicitly identifying the artifacts affected by the annotation. However, we do not support tracing fine-grained variability at the level of source code line. Despite the exact source code location of the annotations may be included as part of the information of the variation point (e.g., adding a new field in the* `Artifact` *class of the metamodel presented in Fig. 4), providing such support would make the maintenance of the mapping model more difficult.* A difference with CVL is that variation points in CVL are defined as part of the variability model, while our mapping is independent of the feature model, and this allows to modify the variation point without affecting the variability specification of the features.

Separation of concerns. Separation of concerns refers to the ability to separate feature functionality into cohesive implementations [3], even when features are crosscutting concerns like the `Tags` and the `Comments` features in our example (Fig. 6). Separation of concerns depends on the implementation carried out by the developers [10], which in turn depends on the programming paradigm (*e.g.,* FOP, AOP) or in the design patterns used (↑: implementation independent, ↖: intended by the programming paradigm, ←: implementation dependent, ↙: simulated with tool support, ↓: no support). For most composition-based approaches, separation of concerns is intended, but not for annotation-based approaches [6] in which this separation can be simulated with tool support (*e.g.,* CIDE [21]). *Likewise in composition-based approaches, separation of concerns in our approach also depends on the implementation of developers. However, in contrast to existing combined approaches, to understand the variability of a feature, it is not necessary to look at the artifact implementation because the mapping model explicitly exposes the variability information through the variation points.*

Information hiding. Information hiding is the separation of a module into internal and external parts (*e.g.,* an interface). A module's implementation should only provide its externally visible part independently from the implementation paradigm used (↑: implementation independent interfaces, ↖: composition mechanism dependent, ←: implementation paradigm dependent, ↙: interfaces implementation depend on the developers, ↓: prevents the usage of interfaces). While some composition-based approaches such as frameworks or components technology provide good support for information hiding, other compositional approaches such as FOP or AOP do not [3]. Annotation-based approaches prevent information hiding because of the fine-grained nature of the features [3]. Information hiding in combined approaches depends on the composition mechanism used but normally is weaker than in pure compositional approaches [10]. *Our approach supports*

Table 4
Comparison of SPL implementation strategies for web engineering.

Quality criteria	Composition [3]	Annotations [3]	Generic integration[6]	FeatureCoPP [10]	CVL [12]	Our approach
Feature traceability	↖ naming conventions	↙ tool support	← naming and tool support	← naming and tool support	↑ binding and references	↑ binding and references
Separation of concerns	↖ intended	↙ simulated with tool support	← impl. dependent	↙ simulated with tool support	← impl. dependent	← impl. dependent
Information hiding	↖ comp. mechanism dependent	↓ prevented	← impl. paradigm dependent	← impl. paradigm dependent	↖ comp. mechanism dependent	↑ impl. independent[AL]
Granularity	← coarse- and medium-grained	↖ fine-grained	↑ all levels	↑ all levels	↑ all levels	↑ all levels
Uniformity	↑ enforce common style	↑ enforce common style	↙ enforce different styles	↖ common style	← different styles[AL]	← different styles[AL,VL]
Preplanning effort	↙ new paradigm	↖ no code changes	← code changes	↖ no code changes	↙ new artifacts and models	↙ new artifacts and models
Language independence	↙ host lang. dependent, monolingual	↑ lang. independent, multilingual	← language dependent	↖ lang. independent with plugins	↑ lang. independent, multilingual	↑ lang. independent, multilingual
Variability type	↖ positive, function, plat./env.	← negative, optional	↖ positive, negative, optional, alternative, function, plat./env.	↖ positive, negative, optional, alternative, function, plat./env.	↑ positive, negative, optional, alternative, function, plat./env., and custom	↑ positive, negative, optional, alternative, function, plat./env., and custom
Automation	← comp. mechanism dependent	↖ tool support	↑ tool support and comp. mechanism	↑ tool support and comp. mechanism	↖ tool support	↖ tool support
Maintainability	↖ unique change	↙ several changes	← specific changes	← specific changes	← specific changes	← specific changes
Evolution	← manual for variable artifacts	← manual for variable artifacts	← manual for variable artifacts	← manual for variable artifacts	↖ automatic with ad-hoc algorithms	← manual for variability models
Tool support	← IDE dependent	↑ well-known tools (C preprocessors, CIDE[21],...)	↙ not dedicated tool	↙ not dedicated tool	↖ dedicated tool (vEXgine[35], scaffolding engine[18])	↖ dedicated tool (feature modeling[45], scaffolding engine[18])

↑ very good support, ↖ good support, ← medium support, ↙ poor support, ↓ no support.
[AL] At the architectural level.
[VL] At the variability specification level.

information hiding well due to our AAM model (i.e., the software architecture vision). For compositional features, we assume their functionality is encapsulated in generic modules or artifacts (not necessary in a one-to-one relation) that we can modify, delete, or replace by other modules that implement the same interfaces, but we are unaware of the specific implementation technique (e.g., AOP). So a module of the AAM model could be a JavaScript component, an aspect in AspectJ, or even a web template in HTML. This is a difference with CVL where application artifacts need to be modeled in a MOF-compliant base model. For annotative features, our approach hides, at the architectural level, the internal variability of the modules and explicitly indicates which modules are affected by fine-grained variability. In any case, our approach does not support information hiding when dealing with variability at the code level as in the majority of annotation-based approaches [10].

***Granularity*.** Granularity describes the level on which variability is implemented [3]. Approaches capable of handling finer degree of granularity are desirable over those that only provide coarse-grained variability (↑: all levels, ↖: fine-grained, ←: medium-grained, ↙: coarse-grained, ↓: no support). Compositional approaches only provide coarse-grained or medium-grained variability at the level of components, classes, methods extensions, etc., while annotative approaches support well fine-grained variability at the level of statements, parameters, or expressions [21]. *Similarly to other integrated approaches [6,10], our approach supports all levels of granularity. On the one hand, multilingual annotations encapsulated in annotative variation points (VP_a) support fine-grained variability at the most finer statements, being possible to annotate the same code line with different annotations, that is, our approach does not enforce undisciplined annotations [3]. On the other hand, composite variation points (VP_c) allow handling coarse-grained variability on top of the hierarchical structure (e.g., files, directories, packets,...) where application modules are physically stored.*

***Uniformity*.** Uniformity refers to the principle that all artifacts (annotated or composed) should be encoded and synthesized in a similar manner or style, regardless of the implementation technique [3] (↑: enforce common style, ↖: common style, ←: different styles, ↙: enforce different styles, ↓: no support). On the hand one, both pure compositional and annotative approaches often enforce a common style by defining a set of rules (*e.g.,* preprocessors for annotations or aspects for AOP), which results in a very good support for uniformity. On the other hand, combined approaches [6] enables developers to use different styles at the same time, providing poor uniformity. The combination proposed by FeatureCoPP [10] uses a single encoding similar to preprocessors, but due to the introduction of composition as an additional implementation layer, uniformity is slightly weakened than in preprocessors. *Our approach allows representing all artifacts subject to variation as generic software artifacts in our AAM model (in the same manner than the CVL approach [12] supports any MOF-compliant model at the architectural level). The AAM model supports artifacts encoded in different styles (components, classes, files, templates,...), and therefore our approach provides a poor uniformity compared with a pure composition or annotative approach. However, in contrast to the generic combined approach [6], the AAM model does not enforce specific styles, and thus, a developer can use the same style (e.g., only components and connectors* *if only architectural artifacts are needed). Moreover, regarding the mapping model, the variability of both annotated and composed artifacts are indistinguishable without the information contained in the variation points. An advantage over the CVL approach is that our variation points do not require to specify their operations from its definition, in contrast to CVL where the operation is defined together with the kind of the variation points (e.g., object substitution, fragment replacement,…), preventing uniformity at the variability specification level.*

Preplanning effort. SPL engineering always incurs a certain amount of preplanning in order to be adopted regardless the approach and implementation [3]. While compositional approaches usually require substantial preplanning activities (*e.g.,* change the code base, learn new programming paradigms,…), annotation-based approaches allow introducing annotations to artifacts with lower efforts [3] (*e.g.,* preprocessors are already well-known in some programming languages), as occurs also for combined approaches based mainly on annotations [10] (↑: variability is intended by the programming paradigm not requiring additional language constructs, ↖: do not require changes in the code base, ←: require changes in the code base, ↙: introduce new paradigm or require building new artifacts/models, ↓: new paradigm and artifacts/models). *Our approach requires building the mapping model that includes the specification of the variation points, their bindings to the features in the feature model, and their references to the artifacts; resulting in a high amount of preplanning (similar effort that for the CVL approach). However, note that existing support for specifying feature models is superior than for CVL variability models [4,35] which greatly facilitates the adoption of our approach over the CVL approach.*

Language independence. In order to implement the variability, approaches should be independent from the host language as well as support multiple languages (↑: language independent and multilingual, ↖: language independent with plugins, ←: language dependent, ↙: monolingual, ↓: no support). Most of the composition-based approaches are monolingual, that is, they work exclusively for one language. Particularly, most of them work on Java, such as AHEAD [47], AspectJ [8], or DeltaJ [48], among others. There are exceptions such as FeatureHouse [30], based on FSTComposer [31], that support more than one language, but a plugin or extension is needed for each language. Annotative approaches are usually multilanguage such as C preprocessor, or CIDE [21], as well as the main commercial alternatives such as Gears [49] or pure::variant [50]. In the generic combination [6], the approach itself is independent of the language but it relies on the particular engines used for composition and annotation. For example, on FeatureC [10] they rely on FeatureHouse [30] and a C preprocessor, so they had to develop a FeatureHouse plugin to support C preprocessor annotations on feature-based modules. FeatureCoPP [10] is based solely on C preprocessor and therefore is independent of the language. However, there is still no tooling support to test this approach with a multilingual product line and evaluate how intrusive the annotations are. *Our approach is completely language independent. For composition, we model the variability of the product line using generic units that can be composed themselves by any kind of artifacts, regardless of the language they are written. For annotations, our derivation engine in charge of resolving the variability does not need any kind of adaptor or plugin, thanks to our multilanguage annotations that can be applied to any application artifact.*

Variability type. An SPL approach supports different types of variability categorized as [51]: positive (functionality is added), negative (functionality is removed), optional (code is included), alternative (code is replaced), function (functionality changes), and platform/environment changes. Here it is better to support more type of variability (↑: support all types of variability including custom variability types, ↖: support more than two variability types, ←: support at least two types of variability, ↙: only support one type of variability, ↓: no support any type of variability). Compositional approaches often support positive, function, and platform/environment variability. Annotative approaches usually support negative and optional variability. Combined approaches based on annotations [6,10] can support also alternative variability. *Our approach supports all types of variability (including negative variability) because we can specify arbitrary operations for the variation points in the mapping model that can be executed by a derivation engine. A difference with CVL is that CVL already provides a larger set of variation points with their predefined operations. However, the operations of the CVL variation points is only based on model transformations, while our operations can be specified as model transformations, executable code scripts, or as direct operations over the file system.*

Automation. The degree of automation is a measure that compares the software elements (*e.g.,* number of components, lines of code) that are manually defined with those that are automatically generated [52,53]. To achieve some degree of automation, compositional approaches rely on their own compositional mechanisms (*e.g.,* weaving aspects for AOP), while annotative approaches are based in some kind of tool support such a preprocessor or a specific tool like CIDE [21]) (↑: tool support and composition mechanism dependent, ↖: tool support, ←: composition mechanism dependent, ↙: ad-hoc mechanism, ↓: no support). This metric can be quantified to allow discussing about the development effort, degree of reuse, and productivity of applying a specific approach. The degree of automation of our approach is defined as the comparison between the number of artifacts (*i.e.,* components, code artifacts, templates, files,…) automatically generated when the variability is resolved ($\#e_a$) and the number of artifacts manually defined ($\#e_m$) to resolve the variability of a specific product [54]:

$$\textsc{Degree of Automation} = \frac{\#e_a}{\#e_a + \#e_m} \tag{1}$$

As shown in Table 3, we observe that the blog SPL using our combined approach contains 117 files, where 67 are variable or contain some annotations, and the remaining 50 files are non-variable or do not need to be modified to derive a product. Without an SPL approach, all the 117 files need to be manually defined, and the 67 variable files need to be manually customized for each product. *Whereas applying an SPL approach as presented in this paper we automatically resolve the variability of those variable files. Concretely, for the blog SPL, we obtain a degree of automation of* $\frac{67}{67+50} = 57\%$. *However, this metric is not specific to our approach, and a similar degree of automation is achieved by any other SPL approach. Indeed, this metric is more suitable to compare specific SPLs and makes sense when considering the developers' efforts when applying, maintaining, and evolving a specific SPL (see the following quality criteria).*

Maintainability. Maintainability is the ease with which a software product can be modified. Maintenance in SPL is more complex because changes in a module can affect various products. Here we are interested in the maintainability of the SPL instead of the individual generated product after delivery. So, the main goal is to evaluate the impact of maintaining the artifacts of the features that compose the SPL [55]. Maintenance is also related to preplanning effort and evolution (↑: a modification does not affect any existing artifact, ↖: a modification only affects a unique artifact, ←: a modification only affects specific well-identified artifacts, ↙: a modification implies several changes in multiple artifacts, ↓: no maintainable or require re-engineering the solution). In this scenario, composition approaches are superior since they promise improvements during maintenance and evolution because of modularity and separation of concerns, requiring to deal with a lesser number of artifacts when modifying them. We observe in Table 3 that in our previous purely annotations solution, developers need to manage 450 annotations scattered among 100 artifacts (files). *In contrast, in our approach, annotations are reduced up to 362 (20% fewer annotations) scattered among 67 artifacts (files), reducing the annotated artifacts to be managed up to 67% of the original ones. This large reduction in the number of annotated artifacts happens because many annotations affect complete files to handle coarse-grained variability, and with the new approach, these annotations have been modeled as compositional variation points* (VP_c). *Nevertheless, in our approach the information of the mapping model needs*

Table 5
Comparison of the CVL approach and our approach using feature models.

Characteristic	CVL approach [12]	Our approach with FMs	Discussion
Variability modeling constructs	VSpecs, Choices, Variables, VClassifiers, Composite VSpecs	Features, Optional/Mandatory features, Feature groups (or, xor), Clonable features (Multi-features), Feature attributes	While the term "feature" is well-known in the SPL community and used in nearly all variability modeling languages, CVL introduces a new dreadful nomenclature of the modeling concepts, despite that the semantics are the same [13].
Cross-tree constraints modeling	Object Constraint Language (OCL)	Propositional Logic (PL)	PL is a simple mathematical standard and can be complemented with higher-order logic if needed. OCL in contrast, is simply too complicated and is tied to MDD and UML [57].
Variation points	Pre-defined (ObjectExistence, LinkExistence, ParametricSlotAssignment,…) and Opaque Variation Points (OVPs)	Custom variation points (Existence, Assignment, Substitution,…)	CVL variation points are all compositional, while our custom variation points can be compositional or annotative.
Base model	MOF-compliant models	AAM models	CVL supports any architectural elements defined according to a MOF-compliant model (*e.g.,* UML). Our AAM model supports arbitrary artifacts with different abstraction such as a component in an architectural model, a class in a class diagram model, a source code file, a database, or a web resource.
Product derivation	M2M transformations (QVT, ATL)	M2M transformations, Scaffolding derivation, code scripts, CLI operations	The operations of the CVL variation points is only based on model transformations over a MOF-compliant model, while our operations can be specified as model transformations, executable code scripts, or even as direct actions over the file system supporting to work with any kind of web artifact (HTML, CSS, Java classes, Python code,…).
Tool support	MoSIS CVL, CVL 2, BVR, KCVL, vEXgine	FeatureIDE, pure::variants, Glencoe, FaMa, Clafer, SPLOT	Almost all of the CVL tools are currently deprecated and even unavailable (*e.g.,* MoSIS CVL, CVL 2, BVR), or abandoned (*e.g.,* KCVL, vEXgine) [35]. However, feature models have the support of the SPL community being the de-facto standard for modeling variability with many tools supporting the different SPL phases [45].

to be maintained, which in fact may suppose a considerable effort. However, the mapping model itself helps to identify and locate the modifications to be performed over the artifacts when a feature is modified. Moreover, the mapping model may be maintained with the help of dedicated tool support as it occurs with the evolution of the CVL models in [56].

Evolution. Evolution is the ability to modify the SPL to support changes, for example, to incorporate new features or functionalities to the SPL. Normally, independently of the SPL approach, this is a manual task to be performed, and practitioners need to define their own ad-hoc algorithms to evolve the variability model and the associated variation points, as occurs with the CVL approach [56] (↑: completely automatic for variable artifacts and variability models, ↖: automatic with ad-hoc algorithms or tool support for variability models, ←: manual for variable artifacts, ↙: manual for variability models, ↓: none support for evolution). *Our approach requires to manually modify the feature model, the AAM model, and the mapping model in order to modify or incorporate new features. However, as occurs with the CVL approach, specific algorithms to automatically evolve those models can be defined [56].*

Tool support. SPL approaches are viable and useful to the extent that they are supported by appropriate tools (↑: there exist several well-known tools that provide good support, ↖: dedicated tool support, ←: IDE dependent tool support, ↙: not dedicated tool but it can be still supported with CASE tools such as common text editors, ↓: no tool support). First, composition-based approaches are supported by tools that depend on the implementation mechanisms. For instance, FeatureIDE [43] for FOP, and AspectJ [8] or AspectC/C++ [58,59] for AOP. These tools are often language and IDE-dependent. Second, most of the annotation-based approaches are supported by C preprocessor, but there is also some specific tool to manage annotations such as CIDE [21]. Finally, the combined approaches have not a dedicated tool and the developer usually needs one or more tools to support both compositional and annotative approaches. *Our approach is supported by a hybrid engine in charge of resolving the variability with the information provided in the mapping model. The hybrid engine is implemented as a CLI tool with Node.js, since it facilitates the invocation of the concrete engines: the scaffolding-based derivation engine [18] to resolve annotative variability and shell scripting to resolve the compositional variability (see Section 3). Despite other engines could be used (ATL engine, CVL engine, C Preprocessor), it requires manually integrating them into the hybrid engine to support them, which can be a considerable implementation effort. Another limitation of the hybrid engine is that it requires specific inputs (feature model, AAM, mapping model, configuration) that, in this case, we specify in JSON format. However, in contrast to CVL which lacks of tool support, feature models are widely supported by several tools in the SPL community [45] as they become the de-facto standard for modeling variability [4].*

5.2.1. Comparison between CVL and feature model-based approaches

An explicit comparison between the CVL approach and our approach using feature models is presented in Table 5. Since the difference between CVL and feature models relies on the variability specification, that is, in the variability model and in the mapping model, the benefits of both approaches are similar as qualitatively shown in Table 4. The quantitative evaluation presented in Section 5.1 is not affected by the usage of feature models in contrast to CVL because the benefits of both approaches over the pure annotation-based approach are the same. Nevertheless, Table 5 summarized the differences of both

CVL and feature model approaches, comparing the different aspects that effectively affect the use of CVL or feature models. Some of these differences have been already discussed throughout the paper.

5.2.2. Adoption of the combined approach

Finally, we would like to briefly discuss the opportunity and adoption challenges of our approach. The case study we have used to present our proposal consists of migrating an annotative-based SPL into a combined approach, which provides a good support for traceability, but requires a complex mapping model. The set of transformation patterns we have identified in Table 2 helps application architects and developers in the migration process and in the definition of the mapping model. The mapping model serves as an additional documentation facet of the SPL. From an architectural point of view, the mapping model facilitates the location of features in software artifacts, improving traceability. Traceability is achieved because the variation points in the mapping model explicitly associate features in the variability model and artifacts in the AAM model, which also helps developers to identify relevant features and artifacts during the development and maintenance of the SPL. The mapping is also used as an input for the hybrid engine to invoke the concrete engines in charge of deriving a product according to the information provided in the variation points. In addition, our proposal can be also applied to an SPL from scratch following any methodology for software development. In this case, to facilitate the adoption of the approach, and to simplify the process of building the mapping model, we consider that using an iterative approach guided by the features could be suitable. That is, each feature is developed in an iteration, and both the product line architecture and the mapping model are updated accordingly; in contrast to first implementing all features and then building the mapping model.

6. Conclusions and future work

We have presented a combined approach to model multiple granular variability based on feature modeling that integrates annotations into a generic composition-based strategy. Our approach effectively handles both fine and coarse-grained variability presented in web applications. The mapping between the feature model and the web artifacts promotes the traceability of the features and the uniformity of the variation points regardless of the granularity of the final source files. The proposed solution uses feature models to specify the variability, in contrast to the CVL language, facilitating the adoption of our approach, while maintaining a good support for feature traceability for both compositional and annotative variation points, at the architectural level.

Our ongoing work considers evaluating our approach with several real-word SPLs in the web engineering domain as well as improving the tooling support by integrating the whole process of modeling and resolution of the variation points in a unique web interface application. We also plan to automate the process of identifying the variation points to reduce the effort of the developers in defining the mapping model.

CRediT authorship contribution statement

Jose-Miguel Horcas: Conceptualization, Methodology, Software, Writing – review & editing. **Alejandro Cortiñas:** Investigation, Software, Writing – review & editing. **Lidia Fuentes:** Work idea, Funding, Supervision, Resources, Project administration. **Miguel R. Luaces:** Validation, Supervision.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.infsof.2022.106910.

Acknowledgments

The work of the authors from the Universidad de Málaga is supported by the projects *Magic* P12-TIC1814 (post-doctoral research grant), *MEDEA* RTI2018-099213-B-I00 (co-financed by FEDER funds), *Rhea* P18-FR-1081 (MCI/AEI/FEDER, UE), *LEIA* UMA18-FEDERIA-157, *TASOVA* MCIU-AEI TIN2017-90644-REDT and, European Union's H2020 research and innovation program under grant agreement DAEMON 101017109. The work of the authors from the Universidade da Coruña has been funded by MCIN/AEI/10.13039/501100011033, NextGenerationEU/PRTR, FLATCITY-POC: PDC2021-121239-C31; MCIN/AEI/10.13039/501100011033 EXTRACompact: PID2020-114635 RB-I00; GAIN/Xunta de Galicia/ERDF CEDCOVID: COV20/00604; Xunta de Galicia/FEDER-UE GRC: ED431C 2021/53; MICIU/FEDER-UE BIZDEVOPSGLOBAL: RTI-2018-098309-B-C32; MCIN/AEI/10.13039/50110 0011033 MAGIST: PID2019-105221RB-C41.

References

[1] M. Marques, J. Simmonds, P.O. Rossel, M.C. Bastarrica, Software product line evolution: A systematic literature review, Inf. Softw. Technol. 105 (2019) 190–208, http://dx.doi.org/10.1016/j.infsof.2018.08.014.

[2] M.A. Laguna, Y. Crespo, A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring, Sci. Comput. Program. 78 (8) (2013) 1010–1034, http://dx.doi.org/10.1016/j.scico.2012.05.003.

[3] S. Apel, D.S. Batory, C. Kästner, G. Saake, Feature-Oriented Software Product Lines - Concepts and Implementation, Springer, 2013, http://dx.doi.org/10.1007/978-3-642-37521-7.

[4] M. Raatikainen, J. Tiihonen, T. Männistö, Software product lines and variability modeling: A tertiary study, J. Syst. Softw. 149 (2019) 485–510, http://dx.doi.org/10.1016/j.jss.2018.12.027.

[5] J. Krüger, M. Pinnecke, A. Kenner, C. Kruczek, F. Benduhn, T. Leich, G. Saake, Composing annotations without regret? Practical experiences using featurec, Softw. Pract. Exp. 48 (3) (2018) 402–427, http://dx.doi.org/10.1002/spe.2525.

[6] C. Kästner, S. Apel, Integrating compositional and annotative approaches for product line engineering, in: Proc. GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering, 2008, pp. 35–40.

[7] C. Prehofer, Feature-oriented programming: A fresh look at objects, in: M. Akşit, S. Matsuoka (Eds.), ECOOP'97 — Object-Oriented Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 1997, pp. 419–443.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: European Conference on Object-Oriented Programming, Springer, 1997, pp. 220–242.

[9] I. Schaefer, L. Bettini, F. Damiani, N. Tanzarella, Delta-oriented programming of software product lines, in: Proceedings of the 14th International Conference on Software Product Lines: Going beyond, in: SPLC'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 77–91, URL http://dl.acm.org/citation.cfm?id=1885639.1885647.

[10] J. Krüger, I. Schröter, A. Kenner, C. Kruczek, T. Leich, FeatureCoPP: compositional annotations, in: C. Seidl, L. Teixeira (Eds.), Proceedings of the 7th International Workshop on Feature-Oriented Software Development, FOSD@SPLASH 2016, Amsterdam, Netherlands, October 30, 2016, ACM, 2016, pp. 74–84, http://dx.doi.org/10.1145/3001867.3001876.

[11] F. Benduhn, R. SCHRöTER, A. Kenner, C. Kruczek, T. Leich, G. ANDSAAKE, Migration from annotation-based to composition-based product lines: towards a tool-driven process, in: Proc. Conf. Advances and Trends in Software Engineering, SOFTENG. IARIA, 2016, pp. 102–109.

[12] J. Horcas, A. Cortiñas, L. Fuentes, M.R. Luaces, Integrating the common variability language with multilanguage annotations for web engineering, in: T. Berger, P. Borba, G. Botterweck, T. Männistö, D. Benavides, S. Nadi, T. Kehrer, R. Rabiser, C. Elsner, M. Mukelabai (Eds.), Proceeedings of the 22nd International Systems and Software Product Line Conference - Vol. 1, SPLC 2018, Gothenburg, Sweden, September 10-14, 2018, ACM, 2018, pp. 196–207, http://dx.doi.org/10.1145/3233027.3233049.

[13] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. ran K. Olsen, A. Svendsen, Adding standardized variability to domain specific languages, in: Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings, IEEE Computer Society, 2008, pp. 139–148, http://dx.doi.org/10.1109/SPLC.2008.25.

[14] T. Berger, P. Collet, Usage scenarios for a common feature modeling language, in: 23rd International Systems and Software Product Line Conference, Vol. B, SPLC 2019, ACM, 2019, pp. 86:1–86:8, http://dx.doi.org/10.1145/3307630.3342403.

[15] V.R. Sánchez, P.N. Ayuso, J.A. Galindo, D. Benavides, Open source adoption factors - a systematic literature review, IEEE Access 8 (2020) 94594–94609, http://dx.doi.org/10.1109/ACCESS.2020.2993248.

[16] C. Hunsen, B. Zhang, J. Siegmund, C. Kästner, O. Leßenich, M. Becker, S. Apel, Preprocessor-based variability in open-source and industrial software systems: An empirical study, Empir. Softw. Eng. 21 (2) (2016) 449–482, http://dx.doi.org/10.1007/s10664-015-9360-1.

[17] A. Cortiñas, M. Luaces, O. Pedreira, A. Places, J. Pérez, Web-based geographic information systems SPLE: Domain analysis and experience report, in: Proceedings of the 21st International Systems and Software Product Line Conference - Volume a, Vol. 1, ACM, Sevilla, Spain, 2017, pp. 190–194, http://dx.doi.org/10.1145/3106195.3106222.

[18] A. Cortiñas, M.R. Luaces, O. Pedreira, A.S. Places, Scaffolding and in-browser generation of web-based GIS applications in a SPL tool, in: M.H. ter Beek, W. Cazzola, O. Díaz, M.L. Rosa, R.E. Lopez-Herrejon, T. Thüm, J. Troya, A.R. Cortés, D. Benavides (Eds.), Proceedings of the 21st International Systems and Software Product Line Conference, SPLC 2017, Volume B, Sevilla, Spain, September 25-29, 2017, ACM, 2017, pp. 46–49, http://dx.doi.org/10.1145/3109729.3109759.

[19] S. El-Sharkawy, N. Yamagishi-Eichler, K. Schmid, Metrics for analyzing variability and its implementation in software product lines: A systematic literature review, Inf. Softw. Technol. 106 (2019) 1–30, http://dx.doi.org/10.1016/j.infsof.2018.08.015.

[20] S. El-Sharkawy, A. Krafczyk, K. Schmid, Fast static analyses of software product lines: an example with more than 42, 000 metrics, in: M. Cordy, M. Acher, D. Beuche, G. Saake (Eds.), VaMoS '20: 14th International Working Conference on Variability Modelling of Software-Intensive Systems, Magdeburg Germany, February 5-7, 2020, ACM, 2020, pp. 8:1–8:9, http://dx.doi.org/10.1145/3377024.3377031.

[21] C. Kästner, S. Apel, M. Kuhlemann, Granularity in software product lines, in: W. Schäfer, M.B. Dwyer, V. Gruhn (Eds.), 30th International Conference on Software Engineering, ICSE 2008, Leipzig, Germany, May 10-18, 2008, ACM, 2008, pp. 311–320, http://dx.doi.org/10.1145/1368088.1368131.

[22] L. Balzerani, D.D. Ruscio, A. Pierantonio, G. De Angelis, A product line architecture for web applications, in: Proceedings of the 2005 ACM Symposium on Applied Computing, in: SAC '05, Association for Computing Machinery, New York, NY, USA, 2005, pp. 1689–1693, http://dx.doi.org/10.1145/1066677.1067059.

[23] M. Laguna, B. González-Baixauli, C. Hernández, Product line development of web systems with conventional tools, in: M. Gaedke, M. Grossniklaus, O. Díaz (Eds.), Web Engineering, ICWE 2009, in: Lecture Notes in Computer Science, vol. 5648, Springer, Berlin, Heidelberg, 2009, pp. 205–212, http://dx.doi.org/10.1007/978-3-642-02818-2_16.

[24] M.A. Naily, M.R.A. Setyautami, R. Muschevici, A. Azurat, A framework for modelling variable microservices as software product lines, in: A. Cerone, M. Roveri (Eds.), Software Engineering and Formal Methods, Springer International Publishing, Cham, 2018, pp. 246–261.

[25] E. Walkingshaw, M. Erwig, A calculus for modeling and implementing variation, in: K. Ostermann, W. Binder (Eds.), Generative Programming and Component Engineering, GPCE'12, Dresden, Germany, September 26-28, 2012, ACM, 2012, pp. 132–140, http://dx.doi.org/10.1145/2371401.2371421.

[26] B. Behringer, Integrating approaches for feature implementation, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, in: FSE 2014, Association for Computing Machinery, New York, NY, USA, 2014, pp. 775–778, http://dx.doi.org/10.1145/2635868.2666605.

[27] B. Behringer, L. Kirsch, S. Rothkugel, Separating features using colored snippet graphs, in: T. Berger, M. Ribeiro (Eds.), Sixth International Workshop on Feature-Oriented Software Development, FOSD '14, VäSterås, Sweden, September 14, 2014, ACM, 2014, pp. 9–16, http://dx.doi.org/10.1145/2660190.2660192.

[28] B. Behringer, S. Rothkugel, Integrating feature-based implementation approaches using a common graph-based representation, in: S. Ossowski (Ed.), Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016, ACM, 2016, pp. 1504–1511, http://dx.doi.org/10.1145/2851613.2851791.

[29] E. Walkingshaw, K. Ostermann, Projectional editing of variational software, in: U.P. Schultz, M. Flatt (Eds.), Generative Programming: Concepts and Experiences, GPCE'14, Vasteras, Sweden, September 15-16, 2014, ACM, 2014, pp. 29–38, http://dx.doi.org/10.1145/2658761.2658766.

[30] S. Apel, C. Kästner, C. Lengauer, Language-independent and automated software composition: The FeatureHouse experience, IEEE Trans. Software Eng. 39 (1) (2013) 63–79, http://dx.doi.org/10.1109/TSE.2011.120.

[31] S. Apel, C. Kästner, Virtual separation of concerns - A second chance for preprocessors, J. Object Technol. 8 (6) (2009) 59–78, http://dx.doi.org/10.5381/jot.2009.8.6.c5.

[32] C. Kästner, S. Apel, M. Kuhlemann, A model of refactoring physically and virtually separated features, in: J.G. Siek, B. Fischer (Eds.), Generative Programming and Component Engineering, 8th International Conference, GPCE 2009, Denver, Colorado, USA, October 4-5, 2009, Proceedings, ACM, 2009, pp. 157–166, http://dx.doi.org/10.1145/1621607.1621632.

[33] D.S. Batory, P. Höfner, B. Möller, A. Zelend, Features, modularity, and variation points, in: A. Classen, N. Siegmund (Eds.), 5th International Workshop on Feature-Oriented Software Development, FOSD '13, Indianapolis, in, USA, October 26, 2013, ACM, 2013, pp. 9–16, http://dx.doi.org/10.1145/2528265.2528269.

[34] D.S. Batory, P. Höfner, D. Köppl, B. Möller, A. Zelend, Structured document algebra in action, in: R.D. Nicola, R. Hennicker (Eds.), Software, Services, and Systems - Essays Dedicated To Martin Wirsing on the Occasion of His Retirement from the Chair of Programming and Software Engineering, in: Lecture Notes in Computer Science, vol. 8950, Springer, 2015, pp. 291–311, http://dx.doi.org/10.1007/978-3-319-15545-6_19.

[35] J.M. Horcas, M. Pinto, L. Fuentes, Extending the common variability language (CVL) engine: A practical tool, in: M.H. ter Beek, W. Cazzola, O. Díaz, M.L. Rosa, R.E. Lopez-Herrejon, T. Thüm, J. Troya, A.R. Cortés, D. Benavides (Eds.), Proceedings of the 21st International Systems and Software Product Line Conference, SPLC 2017, Volume B, Sevilla, Spain, September 25-29, 2017, ACM, 2017, pp. 32–37, http://dx.doi.org/10.1145/3109729.3109749.

[36] T.L. Calvar, F. Jouault, F. Chhel, M. Clavreul, Efficient ATL incremental transformations, J. Object Technol. 18 (3) (2019) 2:1–17, http://dx.doi.org/10.5381/jot.2019.18.3.a2.

[37] L. Burgueño, J. Cabot, S. Gérard, The future of model transformation languages: An open community discussion, J. Object Technol. 18 (3) (2019) 7:1–11, http://dx.doi.org/10.5381/jot.2019.18.3.a7.

[38] Ø. Haugen, O. Øgård, BVR - better variability results, in: D. Amyot, P.F. i Casas, G. Mussbacher (Eds.), System Analysis and Modeling: Models and Reusability - 8th International Conference, SAM 2014, Valencia, Spain, September 29-30, 2014. Proceedings, in: Lecture Notes in Computer Science, vol. 8769, Springer, 2014, pp. 1–15, http://dx.doi.org/10.1007/978-3-319-11743-0_1.

[39] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, A.S. Peterson, Feature-oriented domain analysis (FODA) feasibility study, Technical Report CMU/SEI-90-TR-21, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[40] T. Arendt, E. Biermann, S. Jurack, C. Krause, G. Taentzer, Henshin: Advanced concepts and tools for in-place EMF model transformations, in: 13th International Conference on Model Driven Engineering Languages and Systems, MODELS 2010, in: Lecture Notes in Computer Science, 6394, Springer, 2010, pp. 121–135, http://dx.doi.org/10.1007/978-3-642-16145-2_9.

[41] D.S. Kolovos, R.F. Paige, F. Polack, The epsilon transformation language, in: A. Vallecillo, J. Gray, A. Pierantonio (Eds.), First International Conference on Theory and Practice of Model Transformations, ICMT 2008, in: Lecture Notes in Computer Science, vol. 5063, Springer, 2008, pp. 46–60, http://dx.doi.org/10.1007/978-3-540-69927-9_4.

[42] N. Loughran, P. Sánchez, A. Garcia, L. Fuentes, Language support for managing variability in architectural models, in: C. Pautasso, E. Tanter (Eds.), Software Composition - 7th International Symposium, SC@ETAPS 2008, Budapest, Hungary, March 29-30, 2008. Proceedings, in: Lecture Notes in Computer Science, vol. 4954, Springer, 2008, pp. 36–51, http://dx.doi.org/10.1007/978-3-540-78789-1_3.

[43] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, T. Leich, FeatureIDE: An extensible framework for feature-oriented software development, Sci. Comput. Program. 79 (2014) 70–85, http://dx.doi.org/10.1016/j.scico.2012.06.002, URL http://www.sciencedirect.com/science/article/pii/S0167642312001128, Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010).

[44] A. Joshi, S. Kale, S. Chandel, D.K. Pal, Likert scale: Explored and explained, Br. J. Appl. Sci. Technol. 7 (4) (2015) 396.

[45] J. Horcas, M. Pinto, L. Fuentes, Software product line engineering: a practical experience, in: 23rd International Systems and Software Product Line Conference, Vol. A, SPLC, ACM, 2019, pp. 25:1–25:13, http://dx.doi.org/10.1145/3336294.3336304.

[46] F. Heidenreich, J. Kopcsek, C. Wende, Featuremapper: Mapping features to models, in: Companion of the 30th International Conference on Software Engineering, in: ICSE Companion '08, ACM, New York, NY, USA, 2008, pp. 943–944, http://dx.doi.org/10.1145/1370175.1370199, URL http://doi.acm.org/10.1145/1370175.1370199.

[47] D.S. Batory, J.N. Sarvela, A. Rauschmayer, Scaling step-wise refinement, IEEE Trans. Software Eng. 30 (6) (2004) 355–371, http://dx.doi.org/10.1109/TSE.2004.23.

[48] J. Koscielny, S. Holthusen, I. Schaefer, S. Schulze, L. Bettini, F. Damiani, Deltaj 1.5: Delta-oriented programming for java 1.5, in: Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, in: PPPJ '14, ACM, New York, NY, USA, 2014, pp. 63–74, http://dx.doi.org/10.1145/2647508.2647512, URL http://doi.acm.org/10.1145/2647508.2647512.

[49] C. Krueger, P. Clements, Feature-based systems and software product line engineering with gears from BigLever, in: Proceedings of the 22Nd International Systems and Software Product Line Conference - Vol. 2, in: SPLC '18, ACM, New York, NY, USA, 2018, pp. 1–4, http://dx.doi.org/10.1145/3236405.3236409, URL http://doi.acm.org/10.1145/3236405.3236409.

[50] D. Beuche, Using pure: Variants across the product line lifecycle, in: Proceedings of the 20th International Systems and Software Product Line Conference, in: SPLC '16, ACM, New York, NY, USA, 2016, pp. 333–336, http://dx.doi.org/10.1145/2934466.2962729, URL http://doi.acm.org/10.1145/2934466.2962729.

[51] C. Gacek, M. Anastasopoules, Implementing product line variabilities, SIGSOFT Softw. Eng. Notes 26 (3) (2001) 109–117, http://dx.doi.org/10.1145/379377.375269, URL http://doi.acm.org/10.1145/379377.375269.

[52] R. Lence, L. Fuentes, M. Pinto, Quality attributes and variability in AO-ADL software architectures, in: W. Hasselbring, V. Gruhn (Eds.), Software Architecture, 5th European Conference, ECSA 2011, Essen, Germany, September 13 - 16, 2011. Companion Volume, in: ACM International Conference Proceeding Series, ACM, 2011, p. 7, http://dx.doi.org/10.1145/2031759.2031768.

[53] A. Harrington, V. Cahill, Model-driven engineering of planning and optimisation algorithms for pervasive computing environments, in: Ninth Annual IEEE International Conference on Pervasive Computing and Communications, PerCom 2011, 21-25 March 2011, Seattle, WA, USA, Proceedings, IEEE, 2011, pp. 172–180, http://dx.doi.org/10.1109/PERCOM.2011.5767582.

[54] J.M. Horcas, M. Pinto, L. Fuentes, An automatic process for weaving functional quality attributes using a software product line approach, J. Syst. Softw. 112 (2016) 78–95, http://dx.doi.org/10.1016/j.jss.2015.11.005.

[55] G. Vale, R. Abílio, A. Freire, H. Costa, Criteria and guidelines to improve software maintainability in software product lines, in: 2015 12th International Conference on Information Technology - New Generations, 2015, pp. 427–432, http://dx.doi.org/10.1109/ITNG.2015.75.

[56] J.M. Horcas, M. Pinto, L. Fuentes, Product line architecture for automatic evolution of multi-tenant applications, in: F. Matthes, J. Mendling, S. Rinderle-Ma (Eds.), 20th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2016, Vienna, Austria, September 5-9, 2016, IEEE Computer Society, 2016, pp. 1–10, http://dx.doi.org/10.1109/EDOC.2016.7579384.

[57] D.S. Batory, Should future variability modeling languages express constraints in ocl? in: C. Cetina, O. Díaz, L. Duchien, M. Huchard, R. Rabiser, C. Salinesi, C. Seidl, X. Tërnava, L. Teixeira, T. Thüm, T. Ziadi (Eds.), 23rd International Systems and Software Product Line Conference (SPLC 2019), Vol. B, ACM, 2019, p. 87:1, http://dx.doi.org/10.1145/3307630.3342406.

[58] Y. Coady, G. Kiczales, M. Feeley, G. Smolyn, Using aspectc to improve the modularity of path-specific customization in operating system code, in: ACM SIGSOFT Software Engineering Notes, 26, ACM, 2001, pp. 88–98.

[59] O. Spinczyk, D. Lohmann, M. Urban, Aspectc++: an AOP extension for c++, Softw. Dev. J. 5 (68–76) (2005).