Integrating the Common Variability Language with Multilanguage Annotations for Web Engineering

Jose-Miguel Horcas
Universidad de Málaga, CAOSD Group
Málaga, Spain
horcas@lcc.uma.es

Alejandro Cortiñas
Universidade da Coruña, Laboratorio de Bases de Datos
A Coruña, Spain
alejandro.cortinas@udc.es

Lidia Fuentes
Universidad de Málaga, CAOSD Group
Málaga, Spain
lff@lcc.uma.es

Miguel R. Luaces
Universidade da Coruña, Laboratorio de Bases de Datos
Enxenio S.L.
A Coruña, Spain
luaces@udc.es

ABSTRACT

Web applications development involves managing a high diversity of files and resources like code, pages or style sheets, implemented in different languages. To deal with the automatic generation of custom-made configurations of web applications, industry usually adopts annotation-based approaches even though the majority of studies encourage the use of composition-based approaches to implement Software Product Lines. Recent work tries to combine both approaches to get the complementary benefits. However, technolog-ical companies are reticent to adopt new development paradigms such as feature-oriented programming or aspect-oriented program-ming. Moreover, it is extremely difficult, or even impossible, to apply these programming models to web applications, mainly because of their multilingual nature, since their development involves multiple types of source code (Java, Groovy, JavaScript), templates (HTML, Markdown, XML), style sheet files (CSS and its variants, such as SCSS), and other files (JSON, YML, shell scripts). We propose to use the Common Variability Language as a composition-based ap-proach and integrate annotations to manage fine grained variability of a Software Product Line for web applications. In this paper, we (i) show that existing composition and annotation-based approaches, including some well-known combinations, are not appropriate to model and implement the variability of web applications; and (ii) present a combined approach that effectively integrates annotations into a composition-based approach for web applications. We implement our approach and show its applicability with an industrial real-world system.

KEYWORDS

Automation, annotations, composition, CVL, SPL, variability, web engineering

1 INTRODUCTION

Web applications development involves managing commonality and variability spread over a high diversity of files and resources like code, pages or style sheets, implemented in different languages. To deal with the automatic generation of custom made configurations of web applications, industry usually adopts annotation-based approaches [25, 30] despite the fact that the majority of studies encourage the use of composition-based approaches [11, 28] to implement Software Product Lines (SPLs) [2]. This is mainly because annotations [2, 30] are simple, flexible, and easy to adopt since they are natively supported by many programming languages. In contrast, composition approaches improve modularization, separation of concerns, and maintenance [2]. However, existing composition-based approaches, such as feature-oriented programming (FOP) [40] or aspect-oriented programming (AOP) [32], lack expressiveness and require that industry takes risks and puts high efforts to successfully adopt these new technologies [30, 34].

Moreover, the multilingual nature of web applications, involving multiple types of source code (Java, Groovy, JavaScript), templates (HTML, Markdown, XML), style sheet files (CSS and its variants, such as SCSS), and other kinds of files (JSON, YML, shell scripts), makes extremely difficult, or even impossible to apply some advanced programming models (e.g., FOP, AOP). Besides, web applications must handle great amounts of fine-grained variability,

which can be easily implemented with annotations, but that it would nevertheless be virtually impossible to implement with a composition-based approach. Therefore, in a web developing enterprise like Enxenio[1], it was not practical to adopt a new mechanism that did not use annotations and highly difficulties the implementation of the features. Several works [9, 28, 35] try to combine annotative and composition approaches to get their complementary benefits [28]. These works attempt to introduce feature composition into annotation-based approaches [35], or introduce new implementation layers [9, 28], with the goal of bringing composition techniques closer in practice, but as a result they propose complex approaches to be adopted by industry. So, our goal is that Enxenio continues using their annotations, but improving the modularity of the code and the traceability between features, variation points, components and final source files.

In this paper, we propose to integrate annotations into a composition-based approach, contrary to other approaches that extend annotations with composition mechanisms [34, 35]. Concretely, we use the Common Variability Language (CVL) [20] as a composition-based approach and integrate annotations to manage fine-grained variability of an SPL for web applications. We make the following contributions:

- We show that existing composition and annotation-based approaches, including some well-known combinations [28, 35], are not appropriate to model and implement the kind of variability present in web applications.
- We present a combined approach that effectively integrates annotations into a composition-based approach for web applications.
- We evaluate our approach and discuss its quality criteria in comparison with classical and combined solutions for implementing SPLs.

The rest of the paper is structured as follows. Section 2 discusses related work. Section 3 presents our web-based SPL case study and motivates our approach showing the limitations of the existing approaches. Section 4 presents our combined approach using CVL. Section 5 evaluates our approach taking into account different quality criteria and compares it with the existing approaches. Finally, Section 6 concludes the paper and presents future work.

2 RELATED WORK

This section presents related work in the context of the SPL implementation techniques that combines composition and annotative approaches. Figure 1 summarizes these works differentiating the theoretical researches and practical applications.

Although the FeatureC++ approach [4] already unintentionally integrated compositional and annotative approaches, by using C preprocessors (`#ifdef` annotations) [25], it were Kästner and Apel [28] who first formulated the idea of combining both composition and annotative approaches. Kästner and Apel [28] analyze and compare both composition and annotative approaches, separately in detail, and show the benefits of an integrated approach that introduces an additional implementation layer on top of preprocessors. However, they focus on describing only general ideas for a combined approach, and on discussing the resulting characteristics (granularity, traceability, etc.).

[1] http://www.enxenio.es

Walkingshaw and Erwig [49], Batory [6, 7], and Behringer [9, 10] also provide theoretical researches related with the idea of combining composition and annotative approaches. Walkingshaw and Erwig [49] present *compositional choice calculus*, a formal calculus model to unify composition and annotations, and put it into practice [50] by generating editable documents (views) from a variability-aware *abstract syntac tree*. However, this approach depends on the programming language used. Don Batory [6, 7] proposes two algebraic models: the *feature interaction algebra* and the *structured document algebra*. These models formalize the concept of module with variation points, the composition of them and the decomposition of the modules into smaller parts, simulating annotations for Feature-Oriented Software Development (FOSD) [2]. Behringer et al. [9, 10] propose to unify composition and annotative approaches with adapted tools [10]. In particular, they propose *structured document graphs* [10] based on the *compositional choice calculus* [49] to change between composition, annotations, and the combination of both approaches in an SPL.

Thenceforth, existing work [11, 29, 31, 34, 35] that put into practice the combined approach in SPL are mainly based on the idea of the integrated (hybrid) approach proposed by Kästner and Apel [28]. They claim that the integration is straightforward, conceptually and technically, since it is based on combining existing implementation techniques such as combining preprocessors [25] or virtual separation of concerns [29] with FOP [40], AOP [32], or delta-oriented programming [41]. However, the advantages of these approaches (e.g., modularity, expressiveness,. . .) mainly depend on the specific composition and annotative implementation techniques used. But the election of the programming model (e.g., using or not using AspectJ) should not be imposed by the SPL implementation mechanism. In contrast, we propose a composition-based approach with CVL at the architectural and design level, independently of the SPL implementation technique, that integrates the code level annotations within CVL.

All these approaches [11, 29, 31, 34, 35] are useful in the scenarios of refactoring annotated SPL in order to utilize, or to migrate toward, composition; and to adopt SPLs from legacy systems (the extractive approach) [2]. In particular, Benduhn et al. [11] apply the integration approach proposed by Kästner and Apel [28] in a real case study, by migrating Berkeley DB from C preprocessors annotations toward 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. [35] present FeatureCoPP (Feature Compositional PreProcessor), an integrated implementation 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. In [34], the migration process from annotation-based toward composition-based approaches is applied to the Berkeley DB case study. In contrast to them [34, 35], we propose just the contrary with CVL, that is, we introduce annotations into a composition-based approach.

The conclusion is that technological companies are reluctant to embrace these kinds of combined approaches, mainly when these approaches imposed the adoption of a new programming paradigm. This is even worse if we need to apply a new programming paradigm

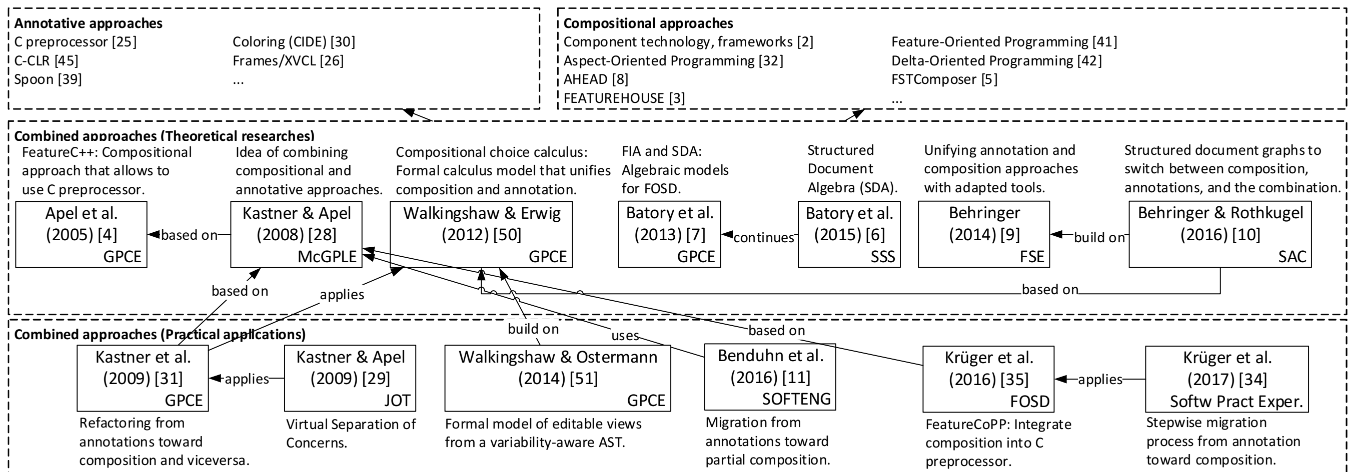

Figure 1: Summary of existing SPL combination techniques.

in the context of web engineering, because we need to use disparate languages and new languages appear on the market every day.

3 MOTIVATION AND CASE STUDY

Enxenio is a small enterprise that has been producing information systems during the last fifteen years for many different domains, including document management systems, academic management systems or business process support systems. One of their particular field of expertise is geographic information systems (GIS) [13, 37, 39]. Independently of the domain, most of their clients demand products built on web. Lately, Enxenio, in collaboration with the Databases Laboratory at University of A Coruña, designed and developed an SPL for web-based GIS [15, 16], with mostly positive results using an annotation-based approach with their own tooling support. However, it is well-known that maintaining and evolving an SPL following this strategy is an arduous task [28, 36, 43, 45], as the number of annotated files grows exponentially and the quality of the annotated source code decreases.

In this section we describe the particular context of Enxenio and the requirements we took into account in order to choose the most appropriate variability implementation technique. Afterwards, we enumerate the limitations found in existing approaches and tools that made them not suitable to meet our needs. To illustrate our points, we propose a simplified example of a web application that covers most of the needs, regarding variability.

3.1 Case study

Since we need to incorporate a Blog into different web applications, we are interested in defining a *Blog* SPL. A blog is a website where entries (called *post*) are HTML text written by registered users of the blog using a post editor. The blog 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 or referenced using a URL. Besides having an author and a timestamp, posts can also be linked with specific tags. In order 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 in order 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 mean to debug the code properly, we can choose to add some extra logging in the code (i.e., a logger).

3.2 Requirements

Web development nowadays involves a high number of different technologies and languages. HTML, CSS and JavaScript are the three standard languages that can be interpreted by any web browser rendering a webpage. However, developers usually use additional programming languages, for instance server-side web technologies such as Java, Python or Ruby, to implement the web application functionality (e.g., connect to a database). Concretely, the code of our Blog SPL involves using a total of 12 different languages or file types, each one including some variable parts: (i) Java and JavaScript programming languages; (ii) some markup languages such as HTML, XML and Markdown; (iii) some style sheet languages such as CSS and SCSS; (iv) some data serialization languages such as YAML and JSON; and, (v) some others standard file formats such as property files or script files. Therefore, it is required that our SPL solution can handle this set of languages, but also any other new language that may appear in the future.

Req1. *The approach needs to be independent of the language, that is, a multilingual SPL approach.*

As previously discussed, web applications expose a high degree of fine-grained variability that usually affects most of the web artifacts specified in different languages. For instance, in the development of web applications it is very common to use many third-party libraries. Each library has its own particular way of being deployed and used, and it may involve, for example, adding some lines in HTML, or defining a parameter in a configuration file. In this case, annotating the lines that vary would be more efficient and easy to understand than modeling this variability with a component.

Req2. *The approach needs to support fine-grained variability.*

In order to evolve and maintain the source code of our SPL, it is desirable to maintain each piece of code implementing a feature,

separately from the rest of code. Also, we would like to easily identify and relate any feature with the final code that implements it, so that the developer can easily add or modify features or code variants maintaining the consistency of the SPL. Indeed, this is the main motivation of our work, to improve the evolution management and maintenance of very large and complex SPLs.

Req3. *The approach must keep the feature traceability between all layers of our SPL and preserve the separation of concerns.*

In the majority of application domains, the product automatically generated using an SPL approach does not need to be modified before its market release. In the web domain, this is not possible, because there are some client customizations that must be performed manually, because it is not possible to model them as part of the SPL. One common example is the adaptation of the user interface to the corporative style of the customer company (i.e., logo, colors, messages style,...). Even if we use one of the well-known Content Management System (CMS) [17] such as *Wordpress*, developers need to modify the actual code file of the templates to adapt the style of the view. So, the product code generated by the SPL normally needs some modifications for a specific customer, and thus, a requirement for our Blog SPL is:

Req4. *The generated code should be as clear and simple as possible, without including any additional glue code or artificial mechanisms that affects the code legibility.*

Besides the technical points, one problematic issue when trying to change the methodology of an enterprise to an SPL-based one is the reticence of the development teams [12] and the costs in time to train both the platform engineers and the product engineers. Recently, in Enxenio, we have tried to totally change the development procedures in order to improve the productivity of the company, but with no success. Developers only adopted a minor part of the new methodology that was partially abandoned. With that bad experience in mind, one of the conditions expressed by the manager team in order to adopt the SPL technology for their products was:

Req5. *To keep the development procedures flexible.*

This requirement particularly means: 1) avoiding technological changes only justified as a mean to fit the implementation technique; 2) tooling support independent of the development IDE and of the operating system; 3) keeping the flexibility in the development of the different components; and 4) lower specific formation requirements, since most of the employees at Enxenio are newly graduated, and their formation is focused on the development procedures and the improvement of their programming skills.

3.3 Limitations of existing approaches

Existing approaches, both compositional and annotative approaches, as well as the combined approaches do not completely fulfill the requirements of web engineering [11, 28, 35].

Composition-based approaches do not support fine-grained variability and the generated code is usually full with code implementing artificial methods created to handle the variability and that make the code very hard to understand and maintain. Besides that, composition-based approaches are not able to work independently of the language (e.g., HTML, CSS, Ruby, JSON,...). There are some attempts to make a multi-language approaches such as FeatureHouse [3], but the fact is that every different language requires to introduce a new plugin supporting the new language specificities, increasing the complexity of the product generation, especially when several languages are used in the same product. Even more, if the syntax of a programming language changes, something very common in web development, the plugin stops working until it is re-programmed to support the new syntax. As an example, most of Java 8 artifacts are still unsupported by AHEAD [8] or FeatureHouse [3]. So, Enxenio cannot use those approaches that have to be maintained by third parties.

Annotation-based approaches can work totally independently of the language as text preprocessors. However, tooling support for annotations are not designed to be used specifically within an SPL approach, and they do not bear in mind the separation of concerns principle. This means that the traceability of features and artifact code is not easy. Moreover, it is well known that the maintenance of the platform code using annotations is a nightmare [2].

The existing combined approaches that mixes both compositional and annotative approaches have similar drawbacks. This is the case of the generic combination approach [28], that depends on the actual composition and annotation-based approaches used. Besides that, introducing composition into an annotation-based approach, like in FeatureCoPP [35], suffers the same problem regarding the poor quality of the generated code, modified with artificial methods created only to handle variability. Moreover, how well traceability and separation of concerns is achieved depends on the implemented solution and not on the approach itself (e.g., FOP).

The approach presented in this paper aims to overcome these limitations and also to address the requirements of SPLs for web applications commented above. In Section 5 we discuss, in detail, the quality criteria of existing approaches in comparison with our approach. Next section presents our approach.

4 OUR APPROACH: INTEGRATING COMPOSITION AND ANNOTATIONS IN CVL

This section details our combined approach for composition and annotation variability with the Common Variability Language (CVL). First, we present how CVL works for composition. Second, we introduce our multilingual annotations. Then, we integrate the annotations in the CVL approach. Finally, we apply the approach to our Blog product line.

As shown in Figure 2, CVL specifies, in separate models, the variability that can be applied to a *base model*. The base model is a model in the domain language that can be defined using a MOF-based metamodel (e.g., UML) and normally does not contain any information about variability. In the context of web engineering the base model represents all the artifacts that are part of the SPL of web applications, from code artifacts to templates or style files. CVL specifies the variability information in a separate variability model, similar to traditional feature models [27]. The variability model also defines the points of the base model that are variable and can be modified during the derivation process — i.e., the variation points (VPs). A configuration model (i.e., a selection of features) describes how the variability is resolved to produce a configured product from the base model. CVL relies on its executable engine to automatically derive a product with the variability resolved.

Table 1: Compositional and annotative variation points.

	Variation Point	Type	Semantic
Predefined[1]	ObjectExistence	Compositional	It indicates the existence of an object in the base model.
	LinkExistence	Compositional	It indicates the existence of a link in the base model.
	ObjectSubstitution	Compositional	It substitutes an object with another one in the final product model.
	FragmentSubstitution	Compositional	It substitutes a set of objects and links (a fragment) with another fragment in the base model.
	ParametricSlotAssignment	Compositional	It assigns a new value to a variable.
	…	…	…
Custom	OVP (Existence)	Annotative	It indicates the existence of portions of text in the files related to an existing object of the base model.
	OVP (Assignment)	Annotative	It assigns a new value to an annotated variable in the files related to an existing object of the base model.
	OVP (Uses)	Annotative	In indicates the existence of a portion of code implementing a «uses» link between two components for different features
	…	…	…

[1] The complete taxonomy of variation points predefined in CVL is available in [19].

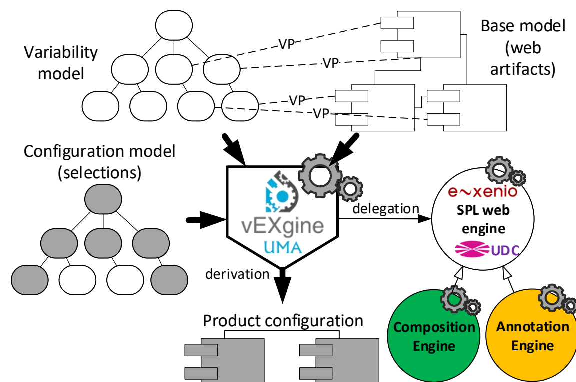

Figure 2: Our approach based on CVL and its tool support.

We propose to extend the CVL approach to allow specifying not only coarse-grained variability, but fine-grained variability that can be resolved with annotations. To bring the CVL approach in practice we use *vEXgine* [24], a customizable and extensible implementation of the CVL execution engine that fully supports the derivation process, including a delegation mechanism that can be extended with different delegation engines. In particular, for this paper, we extend vEXgine to delegate the variability resolution process in a *scaffolding-based derivation engine* adapted from [15] to support the separation of the source code into different components, being each of them composed itself by a set of artifacts. In order to make the integration easier and flexible, we built a web service that receives all the input data from vEXgine, such as the base models and the semantic of the variation points for a specific variability configuration, and generates the source code of the final product. We call this tool *SPL Web Engine*, and it is in charge of resolving the variability of different granularity: (i) coarse-grained and medium-grained variability for those features that can be implemented following a compositional approach (`Composition Engine`); and (ii) fine-grained variability for those features that need to be implemented following an annotative approach (`Annotation Engine`).

4.1 The composition-based approach of CVL

The CVL approach is, by nature, an orthogonal composition-based approach since elements of the base model can be composed, removed, substituted, etc. through the CVL variation points. Variation points specify how the elements of the base models are modified by defining specific modifications to be applied by means of model-to-model (M2M) transformations. The semantics of these transformations are specific to the kind of each variation point. During CVL's execution, the CVL engine (vEXgine in our approach) delegates its control to an M2M engine in charge of executing the transformations specific of each variation point. In our approach, vEXgine delegates its control to a more generic engine (the SPL Web Engine) to execute the semantic of the variation points. Only the semantic of those variation points bound to a selected feature in a configuration model will be executed during variability resolution.

Some of the variation points supported and predefined by CVL for composition 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*), among others (upper part of Table 1). A very powerful variation point is the Opaque Variation Point (OVP) that enables to define a custom-made variation point with a new tailored semantic, not pre-defined in CVL.

4.2 Multilanguage annotations for web SPLs

In this subsection we will present the annotations used in our combined approach. As previously said, in CVL the base model does not contain any information about coarse-grained variability – i.e., this kind of variability is specified in the variability model with the variation points. However, in the context of our SPL for web applications, we do allow that artifacts of the base model contain variability information, but fine-grained variability, implemented by means of annotations. These annotations correspond with the variability that makes no sense to be managed by a composition-based approach (e.g., to model variable parts of a HTML web form).

The annotation engine used by the SPL Web Engine processes each file as plain text. This means that our annotations are language independent or multilanguage since we can annotate any text-based web artifact used in our case study (HTML, CSS, Java, etc.). When an annotation is found, the code associated with this annotation is evaluated. This code is different depending on the kind of variation point (lower part of Table 1), and can define: (i) the existence of annotations in files related to elements of the base model (*Existence*); (ii) the assignment of values through annotations to elements of the base model (*Assignment*); and, (iii) the interaction between two elements from the base model (*Uses*). The annotations are simple JavaScript code embedded in comments (see in Figure 3, `if` sentences for the *Existence* variation point). In Figure 3 we can see simplified excerpts of annotated code for the functionality of the blog associated with the inclusion of images in a post (a) for the view (HTML), and (b) for the controller (JavaScript). In this example, all annotations implement an *Existence* variation point. In the JavaScript code you can also note that if the feature selected is

```html
<div class="btn-group">
    <button type="button" class="btn btn-default"
      ng-click="$ctrl.toggleHTML()" title="Toggle HTML / Markdown">
        <span class="glyphicon glyphicon-open"
      aria-hidden="true"></span>
    </button>
    <!--% if (feature.imageFromURL) { %-->
    <button type="button" class="btn btn-default"
      ng-click="$ctrl.insertImageFromURL()" title="Insert from URL">
        <span class="glyphicon glyphicon-picture"></span>
    </button>
    <!--% } %-->
    <!--% if (feature.imageUploading) { %-->
    <button type="button" class="btn btn-default"
      ng-click="$ctrl.uploadImage()" title="Upload image">
        <span class="glyphicon glyphicon-open"></span>
    </button>
    <!--% } %-->
    <button type="button" class="btn btn-default"
      ng-click="$ctrl.showMarkdownHelp()" title="Show Markdown help">
        <span class="glyphicon glyphicon-open"></span>
    </button>
</div>
```

(a) Example of annotated HTML code.

```javascript
angular.module('blog').component('editor', {
    bindings : { post: '=' },
    controller: controller,
    templateUrl: 'app/components/editor/editor.html'
});
function controller($showdown, /*% if (feature.imageUploading) {
       */FileUploader, /*% } */ModalService) {

    this.toggleHTML = function() { ... };

    /*% if (feature.imageFromURL) { */
    this.insertImageFromURL = function() {
        // implementation of the feature
    };
    /*% } */
    /*% if (feature.imageUploading) { */
    this.uploadImage = function() {
        // implementation of the feature
    };
    /*% } */

    this.showMarkdownHelp = function() { ... };
}
```

(b) Example of annotated JavaScript code.

Figure 3: Example of multilingual annotations for two artifacts in different languages.

imageUploading the annotation adds a new parameter *FileUploader*, which is not present if the other feature *imageFromURL* is selected. This is a good representative example of how fine-grained annotations can be.

Note that the annotations are comments in both of the excerpts shown, and they will not interfere with the source code editors. This is a very simple but really interesting feature of our derivation engine: it allows customizing the delimiters for the annotations depending on the file. This is not done with any extra plugin, but 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 a HTML editor.

4.3 Integrating fine-grained variability in CVL

CVL is intentionally a compositional approach that is applied at a high level of abstraction like the architectural level instead of working at code level. However the CVL approach is unaware of how the features are physically separated into code units implementing the base model (e.g., components, aspects, feature modules), and therefore, any composition-based approach at the code level could be applied, such 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 [28]. So, one code unit can implement a variable feature and at the same time contain variable code text. On the other hand, in web applications sometimes it is not possible to physically separate independent features in different code units, as desirable. Or a developer reluctant to adopting our new approach prefer to use annotations to implement variable features at code level. In these cases, we can use annotations to identify parts of the code implementing different variable features. To handle all these case we propose to introduce annotations (our multilingual annotations) within the CVL approach to be able to handle the fine-grained variability from a high abstract level. This allows CVL to also resolve the variability defined by the annotations, apart from the compositional variability, during the product derivation.

Figure 4 shows an schema of our combined approach with CVL. The variability model consists of two main parts: (1) an abstract level with the feature tree (i.e., VSpec tree in CVL terminology) as in traditional feature models [27], and (2) a concrete level with the variation points (VPs). At the concrete level (i.e., the variation points), it is possible to distinguish the variability granularity that affects each component since variation points refer directly artifacts of the base model that implement each feature. We classify variation points into two categories:

Compositional variation point (VPc). Compositional variation points define the coarse-grained variability that is applied at the architectural level. These are the traditional variation points provided by CVL such as *ObjectExistence*, *LinkExistence*, *FragmentSubstitution*, or *ParametricsSlotAssignment*. The semantic of the VPc is predefined by CVL, but we can also define our own semantic for VPc using Opaque Variation Points (OVPs).

Annotative variation point (VPa). Annotative variation points define the fine-grained variability that is applied at a lower level of abstraction — i.e., at the artifact level such as source code or web templates. An VPa is an Opaque Variation Point (OVP) the semantic of which generically specifies that "there is an annotation bound to the selected feature and the annotation is located in the component/artifact this variation point refers to".

Lower part of Table 1 shows some annotative variation points we define to handle fine-grained variability. From this classification of variation points we can now distinguish the granularity of each feature. First, mandatory features are always present in every generated product, so there is not variability to handle. Secondly, it is possible to include abstract features when necessary for organization purpose (e.g., group optional features), but since they do not represent a concrete variable artifact, they are not bound to any

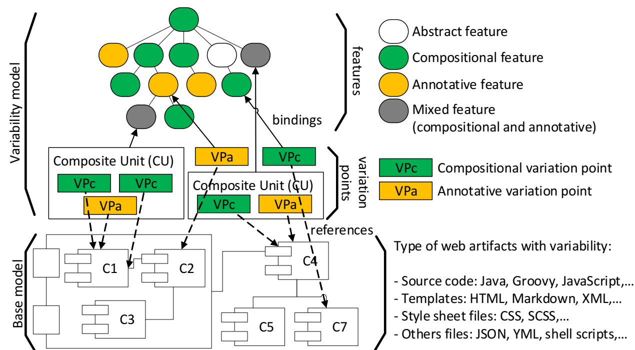

Figure 4: Integrating composition and annotations in CVL.

variation point. Finally, variable features can be implemented as compositional, annotative, or both (mixed), based on the nature of the feature and on the analysis of the developer. A compositional feature is bound to a compositional variation point, and its semantic will be resolved by the `Composition Engine` (see Figure 2). An annotative feature is bound to an annotative variation point, which will be resolved by the `Annotation Engine`. A mixed feature (i.e., a feature that involves composition and annotations) is bound to a set of variation points encapsulated in a *Composite Unit* (*CU*). A CU in CVL is a unit of modularization that contains a set of variation points to jointly handle the variability of a specific feature or set of features. If the mixed feature is selected in a configuration the semantic of all variation points belonging to the bound CU will be processed by the SPL web engine.

Note that it is also possible to encapsulate in a CU, many variation points of the same type (compositional or annotative). This is useful in those cases where the feature implementation is scattered because of a bad design. So, thanks to the use of CVL, our approach can improve the feature traceability of the SPL independently from the implementation technique used at the code level.

4.4 Applying our approach to the Blog SPL

Figure 5 shows how our approach is applied to the Blog SPL. It presents the variability model specified in CVL that includes the two kinds of variation points (compositional and annotative), and the references to the variable artifacts of the base model. Note that in order to simplify the figure we have omitted some links connecting features, variation points and base model.

On the one hand, there are some compositional features that can be grouped in a CU. This is the case of the feature `Comments`, which adds several modules into the final product as specified by the links of each variation point. Each of these variation points is linked to one or several components handling the API of the comments in the server side of the architecture (`CommentREST`), and some client-side components for writing a comment (`CommentEditor`), viewing existing comments (`CommentViewer`) and handling the communication with the API of the comments (`CommentResource`). When the feature `Comments` is selected in a configuration, the final product will include all these modules providing the functionality of managing the comments, while if the feature `Comments` is not present in a configuration, all the related modules will be excluded from the final product.

On the other hand, there are some annotative features that require a finer degree of variability such as the `ImageFromURL` feature (see Figure 3). The annotative variation point associated with this feature includes a reference to the component affected by the annotation. When the `ImageFromURL` feature is selected in a configuration, the derivation engine will resolve the annotation with the semantic specified and will generate the final product's code with the variability resolved and without any annotations left (in 3 (a) lines 5, 9-14 are removed and also in (b) comments of line 9, 13, 17 and lines 19-23). The HTML and Javascript code show respectively one variant of the view and the controller implementations of the `PostEditor` component (implemented using the MVC pattern).

Finally, there are also some mixed features (compositional and annotative) whose implementation is much simpler using annotations but which also requires the inclusion of a component as for example the `ImageUploading` and the `Internationalization` features. For instance, the feature `ImageUploading` can be implemented with barely a few lines added to the JavaScript source code of the different editors (see Figure 3). However, it also requires a generic component able to handle the file uploading in both client (`FileUploaderClient`) and server side (`FileUploaderService`). All the variability of the `ImageUploading` feature is encapsulated in a CU, and thus, when this feature is selected in a configuration, 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. The derivation engine first resolves the compositional variation points, and then the annotative variation points. This way we prevent to resolve fine-grained variability of components that will be not present in the final product.

5 EVALUATION

This section discusses and compares our approach to the pure composition and annotation-based approaches and to the most relevant integrated approaches [28, 35]. We partly base the discussion on the quality criteria for SPL implementation techniques defined in [2] (*feature traceability*, *separation of concerns*, *information hiding*, *granularity*, *uniformity*, and *preplanning effort*) but from a different point of view, that is, the architectural level where CVL works. In addition, 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*. We also compare our approach with a previous solution for web applications used by Enxenio, based only on annotations (Section 3). To do that, we quantitatively assess those quality criteria that can be measured with a metric in a Web-based GIS product line (around 45K LOC) developed by Enxenio [15, 16].

Feature traceability. Feature traceability describes the mapping between a feature in the variability model and its implementation in an artifact or set of artifacts. For compositional approaches, this mapping depends on the implementation technique. It is said [2, 35] 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

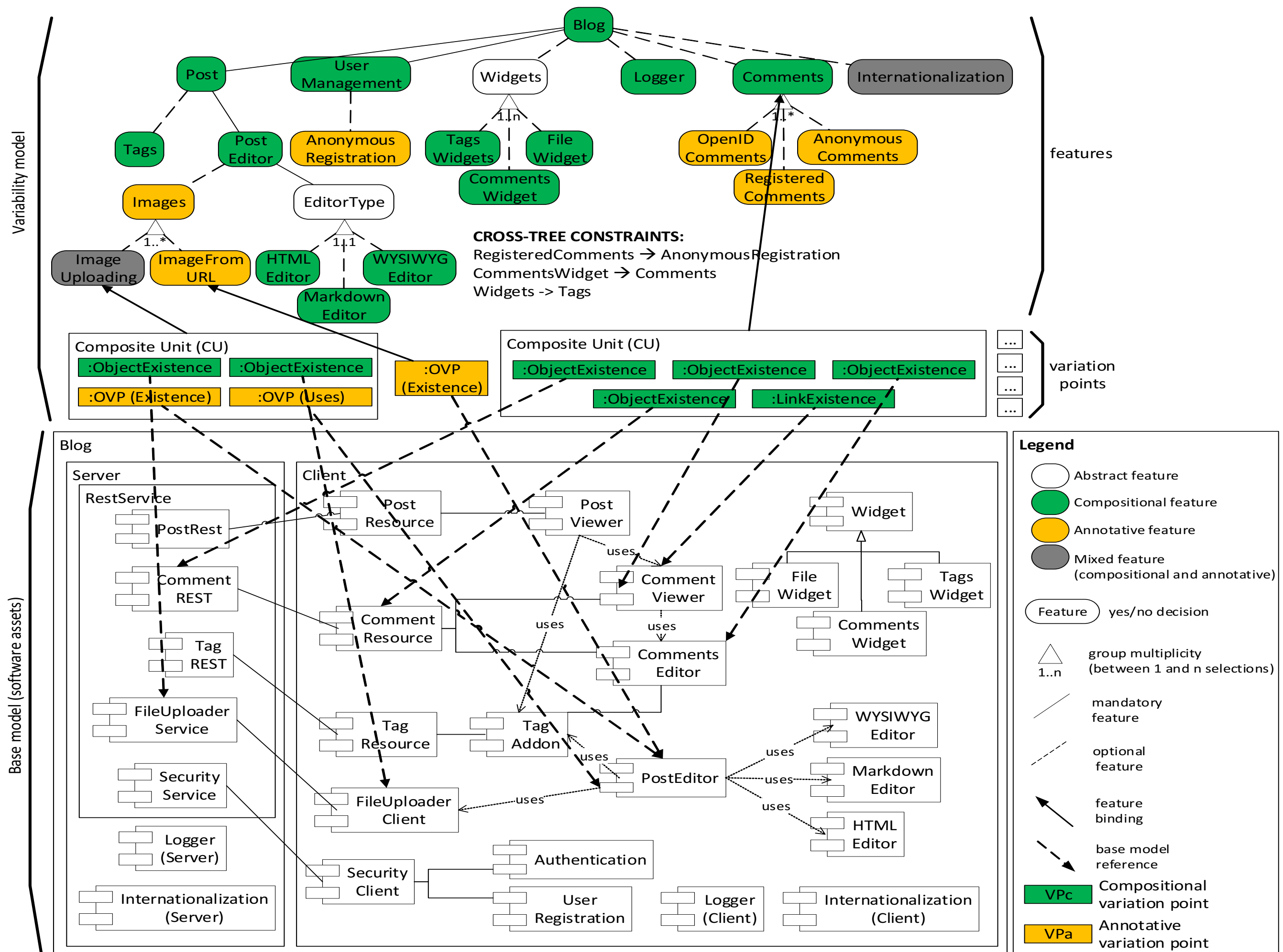

Figure 5: Our approach applied to the Blog product line case study.

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, 30]. 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 [2]. Traceability in combined approaches is weaker than in pure compositional approaches because existing integrating approaches [28, 35] are basically annotative approaches that try to introduce composition. With CVL, our approach allows tracing a feature (compositional or annotative) explicitly from the variability model to the artifacts. This is done thanks to the bindings and references defined by the variation points (see Figure 4) that explicitly bind each feature of the variability model to the target artifacts in the software architecture. 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.

Separation of concerns. Separation of concerns refers to the ability to separate feature functionality into cohesive implementations [2], even when features are crosscutting concerns like the `Logger` and the `Internationalization` features in our example. Separation of concerns depends on the implementation carried out by the developers [35], that in turns depends on the programming paradigm used (e.g., FOP, AOP). For most composition-based approaches, separation of concerns is intended, but not for annotation-based approaches [28] in which this separation can be simulated with tool support (e.g., CIDE [30]).
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 modeling a concern, it is not necessary to look at the artifact implementation because the variability model explicitly exposes the variability information through the variation points.

Information hiding. Information hiding is the separation of a module into internal and external part (e.g., an interface). 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 [2]. Annotation-based approaches prevent information hiding because

of the fine-grained nature of the features [2]. Information hiding in combined approaches depends on the composition mechanism used, but normally is weaker than in pure compositional approaches [35].

Our approach supports information hiding well due to 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 base model could be a JavaScript component, an aspect in AspectJ, or even a web template in HTML. Note that, from the point of view of the information hiding, our approach works as the CORE approach [1, 42] for composition. 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 [35].

Granularity. Granularity describes the level on which variability is implemented [2]. 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 statement, parameters, or expressions [30].

Similarly to other integrated approaches [28, 35], our approach supports all levels of granularity. On the one hand, multilingual annotations 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 [2]. On the other hand, the composition mechanism provided by CVL allows coarse-grained variability on top of the hierarchical structure (e.g., packets, directories,…) 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 the implementation technique [2]. Both pure compositional and annotative approaches often enforce a common style (e.g., preprocessors for annotations or aspects for AOP). But combined approaches [28, 35] enables developers to use different styles at the same time.

Our approach allows representing all artifacts subject to variation as software components of an architectural model (e.g., in UML). The variability of both annotated and composed artifacts are indistinguishable without the information contained in the variation points.

Preplanning effort. SPL engineering always incurs a certain amount of preplanning [2]. While compositional approaches usually require substantial preplanning activities, annotation-based approaches allows introduce annotations to artifacts with lower efforts [2], as occurs also for combined approaches based mainly on annotations [28, 35]. Our approach requires to build the variability model (similar effort that for the feature model) in addition to the specification of the variation points, their bindings to the features and their references to the artifacts; resulting in a high amount of preplanning. However, note that the specification of the variability model including the variation points is done only once at the domain engineering stage of the SPL engineering process.

Multilanguage and language independence. 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 [8], AspectJ [32], or DeltaJ [33], among others. There are exceptions such as FeatureHouse [3], based on FSTComposer [5], that support more than one language, but a plugin or extension is needed for each language.

Although, there are also some annotation-based tools that only work for a specific language (e.g., Spoon [38] for Java), annotative approaches are usually multilanguage such as C preprocessor, CIDE [30], Frames/XVCL [26], or C-CLR [44], as well as the main commercial alternatives such as Gears[2] or pure::variant[3].

Regarding some of the existing alternatives that combine composition and annotations, in the generic combination [28] the approach itself is independent of the language but it relies on the particular engines used for composition and annotation. For example, on FeatureC [35] they rely on FeatureHouse [3] and on C preprocessor, so they had to develop a FeatureHouse plugin to support C preprocessor annotations on feature-based modules. FeatureCoPP [35] 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 UML components that can be composed themselves by any kind of artifacts, regardless of the language they are written. For annotations, our SPL Web Service in charge of resolving the variability does not need any kind of adapter or plugin, thanks to our multilanguage annotations (Section 4.2).

Variability type. An SPL approach supports different types of variability categorized as [18]: positive (functionality is added), negative (functionality is removed), optional (code is included), alternative (code is replaced), function (functionality changes), and platform/environment changes. Compositional approaches often support positive, function, and platform/environment variability. Annotative approaches usually support negative and optional variability. Combined approaches based on annotations [28, 35] can support also alternative variability.

Our approach supports all type of variability thanks to the predefined variation points of CVL (Table 1), but also allows to incorporate new types of variabilities user-defined with the OVPs.

Degree of automation and effort. 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 [22]. This metric allows discussing about the development effort, degree of reuse and productivity of applying a specific approach. We define the degree of automation of our approach as the comparison between the number of elements (i.e., code artifacts, templates, files,…) automatically generated when the variability is resolved ($\#e_a$) and the

[2]http://www.biglever.com/
[3]http://www.pure-systems.com/

Table 2: Quality criteria of our approach in comparison with existing SPL implementation strategies.

Quality Criteria	Composition-based approaches	Annotation-based approaches	Generic combination [28]	FeatureCoPP [35]	**Our Approach**
Feature traceability	naming	tool support	naming and tool support	naming and tool support	bindings and references
Separation of concerns	intended	simulated with tool support	implementation dependent	implementation dependent	implementation dependent
Information hiding	comp. mechanism dependent	prevented	comp. mechanism dependent	comp. mechanism dependent	independent[AL]
Granularity	coarse- and medium-grained	fine-grained	all levels	all levels	all levels
Uniformity	common style	common style	different styles	different styles	common style[AL]
Preplanning effort	high	low	low	low	medium
Language independence and multilanguage	host language dependent, monolingual	language independent	approach dependent	language independent	language 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
Automation	comp. mechanism dependent	tool support	comp. mechanism dependent and tool support	comp. mechanism dependent and tool support	tool support
Maintainability	low	high	medium	medium	medium
Evolution	manual	manual	manual	manual	automatic with algorithms
Tool Support	IDE dependent	C preprocessor, CIDE,…	Not dedicated tool	Not dedicated tool	vEXgine [24], and Enxenio tool [15]

[AL] At the architectural level.

Table 3: SPL for web-based GIS applications.

Metric	Annotative approach	CVL combined approach
#features	128	128
#products (configurations)	255,704	255,704
#artifacts (code files, templates,…)	621	621
#annotated artifacts	504	166
#annotations	2,592	1,694

number of elements manually defined ($\#e_m$) in order to manually resolve the variability of a specific product:

$$\text{Degree of Automation} = \frac{\#e_a}{\#e_a + \#e_m} \quad (1)$$

As shown in Table 3, we observe that the web-based GIS product line contains 621 artifacts, where 504 are variable or contain some degree of variability. This means that to manually generate a specific product we have to manually modify up to 504 different artifacts. Whereas applying an SPL approach as the presented in this paper we automatically resolve the variability of those artifacts, obtaining a degree of automation of 81.16%. However, the same degree of automation is achieved by any other SPL approach, so this metric makes sense when considering the developers' efforts when applying and maintaining an specific SPL.

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 [48].

We observe in Table 3 that in our previous approach, developers needs to manage 2592 annotations scattered among 504 artifacts. In contrast, in our CVL approach, annotations are reduced up to 1694 (35% less annotations) scattered among 166 artifacts, reducing the annotated artifacts to be managed up to 67%. This large reduction in the number of annotated artifacts is due to the fact that many annotations affect complete files to handle coarse-grained variability, and with the new approach these annotations have been modeled in CVL as compositional variation points.

Evolution. Evolution is the ability to modify the SPL to support changes at the domain engineering level, as 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. In contrast, the SPL built on our approach can be automatically evolved applying some evolution algorithms formally defined in [23]. These algorithms allow incorporating changes to the CVL variability model and propagating those changes to the configurations and generated products.

Tool support. SPL approaches are viable and useful to the extent that they are supported by appropriate tools. Composition-based approaches are supported by tools that depend on the implementation mechanisms such as FeatureIDE [47] for FOP, or AspectJ [32] or AspectC/C++ [14, 46] for AOP. These tools are often language and IDE dependent. Most of annotation-based approaches are supported by C preprocessor, but there is also some specific tool to manage annotations such as CIDE [30]. 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 the vEXgine tool [24] and the SPL Web Engine of Enxenio [15] that work in conjunction as explained in Section 4.

Table 2 summarizes and compares the results of our approach with pure composition-based and pure annotation-based approaches and with the two more well-known integration approaches of Kästner and Apel [28] and FeatureCoPP [35].

6 CONCLUSIONS AND FUTURE WORK

We have presented an integrated solution that combines annotations into the composition-based approach of CVL to handle the variability existing in web applications at different levels of granularity. To do so, we have extended the CVL approach and provided appropriate tool support.

Our approach keeps the mapping between the features and their implementation artifacts through the bindings and references from the variation points to the feature model and to the base models, respectively. So we provide uniform and good support for feature traceability for both compositional and annotative strategies, at the architectural level. The proposed solution is completely language independent through the use of MOF-based models at the architectural level for coarse-grained variability and by means of multilanguage annotations for fine-grained variability.

As future work, we plan to evaluate our approach with several SPLs in the web engineering context. In parallel, pursuing a better maintenance support, we need to add analytic features to our tools (e.g., looking for *dead features* or checking consistency of annotations in code linked to OVPs in CVL).

ACKNOWLEDGEMENTS

The work of the authors from the Universidad de Málaga is supported by the projects Magic P12-TIC1814 and HADAS TIN2015-64841-R (co-financed by FEDER funds). The work of the authors from the Universidade da Coruña has been funded by Xunta de Galicia/FEDER-UE CSI: ED431G/01; GRC: ED431C 2017/58. MINECO-AEI/FEDER-UE Datos 4.0: TIN2016-78011-C4-1-R; Flatcity: TIN2016-77158-C4-3-R. EU H2020 MSCA RISE BIRDS: 690941.

REFERENCES

[1] Omar Alam, Jörg Kienzle, and Gunter Mussbacher. 2013. Concern-Oriented Software Design. In *Proceedings of the 16th International Conference on Model-Driven Engineering Languages and Systems - Volume 8107*. Springer-Verlag New York, Inc., New York, NY, USA, 604–621. https://doi.org/10.1007/978-3-642-41533-3_37

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-oriented software product lines*. Springer.

[3] S. Apel, C. Kästner, and C. Lengauer. 2013. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Transactions on Software Engineering* 39, 1 (Jan 2013), 63–79. https://doi.org/10.1109/TSE.2011.120

[4] Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. 2005. FeatureC++: on the symbiosis of feature-oriented and aspect-oriented programming. In *International Conference on Generative Programming and Component Engineering*. Springer, 125–140.

[5] Sven Apel and Christian Lengauer. 2008. Superimposition: A Language-Independent Approach to Software Composition. In *Software Composition*, Cesare Pautasso and Éric Tanter (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 20–35.

[6] Don Batory, Peter Höfner, Dominik Köppl, Bernhard Möller, and Andreas Zelend. 2015. Structured document algebra in action. In *Software, Services, and Systems*. Springer, 291–311.

[7] Don Batory, Peter Höfner, Bernhard Möller, and Andreas Zelend. 2013. Features, Modularity, and Variation Points. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development (FOSD '13)*. ACM, New York, NY, USA, 9–16. https://doi.org/10.1145/2528265.2528269

[8] Don S. Batory, Jacob Neal Sarvela, and Axel Rauschmayer. 2004. Scaling Step-Wise Refinement. *IEEE Trans. Software Eng.* 30, 6 (2004), 355–371. https://doi.org/10.1109/TSE.2004.23

[9] Benjamin Behringer. 2014. Integrating Approaches for Feature Implementation. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 775–778. https://doi.org/10.1145/2635868.2666605

[10] Benjamin Behringer and Steffen Rothkugel. 2016. Integrating Feature-based Implementation Approaches Using a Common Graph-based Representation. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC '16)*. ACM, New York, NY, USA, 1504–1511. https://doi.org/10.1145/2851613.2851791

[11] F Benduhn, R SCHRöTER, A Kenner, C Kruczek, T Leich, and G ANDSAAKE. 2016. 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*. 102–109.

[12] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. A survey of variability modeling in industrial practice. *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems - VaMoS '13* (2013), 1. https://doi.org/10.1145/2430502.2430513

[13] N R Brisaboa, J A Cotelo-Lema, A Fariña, M R Luaces, J R Parama, and J R R Viqueira. 2007. Collecting and publishing large multiscale geographic datasets. *Software: Practice and Experience* 37, 12 (oct 2007), 1319–1348. https://doi.org/10.1002/spe.807

[14] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. 2001. Using AspectC to improve the modularity of path-specific customization in operating system code. In *ACM SIGSOFT Software Engineering Notes*, Vol. 26. ACM, 88–98.

[15] A. Cortiñas, M.R. Luaces, O. Pedreira, and Á.S. Places. 2017. Scaffolding and in-browser generation of web-based GIS applications in a SPL tool. *Proceedings of the 21st International Systems and Software Product Line Conference - Volume B* 2 (2017), 46–49. https://doi.org/10.1145/3109729.3109759

[16] A. Cortiñas, M.R. Luaces, O. Pedreira, Á.S. Places, and J. Pérez. 2017. 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, 190–194. https://doi.org/10.1145/3106195.3106222

[17] V. M. A. d. Lima, R. M. Marcacini, M. H. P. Lima, M. I. Cagnin, and M. A. S. Turine. 2014. A Generation Environment for Front-End Layer in e-Government Content Management Systems. In *2014 9th Latin American Web Congress*. 119–123. https://doi.org/10.1109/LAWeb.2014.20

[18] Critina Gacek and Michalis Anastasopoules. 2001. Implementing Product Line Variabilities. *SIGSOFT Softw. Eng. Notes* 26, 3 (May 2001), 109–117. https://doi.org/10.1145/379377.375269

[19] Øystein Haugen. 2012. Common Variability Language (CVL) OMG Revised Submission. *OMG Doc. ad/2012-08-05* (2012).

[20] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen. 2008. Adding Standardized Variability to Domain Specific Languages. In *2008 12th International Software Product Line Conference*. 139–148. https://doi.org/10.1109/SPLC.2008.25

[21] Florian Heidenreich, Jan Kopcsek, and Christian Wende. 2008. FeatureMapper: Mapping Features to Models. In *Companion of the 30th International Conference on Software Engineering (ICSE Companion '08)*. ACM, New York, NY, USA, 943–944. https://doi.org/10.1145/1370175.1370199

[22] Jose Miguel Horcas, Mónica Pinto, and Lidia Fuentes. 2016. An automatic process for weaving functional quality attributes using a software product line approach. *Journal of Systems and Software* 112 (2016), 78–95. https://doi.org/10.1016/j.jss.2015.11.005

[23] Jose Miguel Horcas, Mónica Pinto, and Lidia Fuentes. 2016. Product Line Architecture for Automatic Evolution of Multi-Tenant Applications. In *20th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2016, Vienna, Austria, September 5-9, 2016*. 1–10. https://doi.org/10.1109/EDOC.2016.7579384

[24] Jose Miguel Horcas, Mónica Pinto, and Lidia Fuentes. 2017. Extending the Common Variability Language (CVL) Engine: A practical tool. In *Proceedings of the 21st International Systems and Software Product Line Conference, SPLC 2017, Volume B, Sevilla, Spain, September 25-29, 2017*. 32–37. https://doi.org/10.1145/3109729.3109749

[25] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßenich, Martin Becker, and Sven Apel. 2016. Preprocessor-based variability in open-source and industrial software systems: An empirical study. *Empirical Software Engineering* 21, 2 (01 Apr 2016), 449–482. https://doi.org/10.1007/s10664-015-9360-1

[26] S. Jarzabek, P. Bassett, Hongyu Zhang, and Weishan Zhang. 2003. XVCL: XML-based variant configuration language. In *25th International Conference on Software Engineering, 2003. Proceedings.* 810–811. https://doi.org/10.1109/ICSE.2003.1201298

[27] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.

[28] Christian Kästner and Sven Apel. 2008. Integrating compositional and annotative approaches for product line engineering. In *Proc. GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering*. 35–40.

[29] Christian Kästner and Sven Apel. 2009. Virtual separation of concerns-a second chance for preprocessors. *Journal of Object Technology* 8, 6 (2009), 59–78.

[30] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2008. Granularity in Software Product Lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 311–320. https://doi.org/10.1145/1368088.1368131

[31] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2009. A Model of Refactoring Physically and Virtually Separated Features. In *Proceedings of the Eighth International Conference on Generative Programming and Component Engineering (GPCE '09)*. ACM, New York, NY, USA, 157–166. https://doi.org/10.1145/1621607.1621632

[32] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *European conference on object-oriented programming*. Springer, 220–242.

[33] Jonathan Koscielny, Sönke Holthusen, Ina Schaefer, Sandro Schulze, Lorenzo Bettini, and Ferruccio Damiani. 2014. 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 (PPPJ '14)*. ACM, New York, NY, USA, 63–74. https://doi.org/10.1145/2647508.2647512

[34] Jacob Krüger, Marcus Pinnecke, Andy Kenner, Christopher Kruczek, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. Composing Annotations Without Regret? Practical Experiences Using FeatureC. *Software Prac. Experience* (2017). https://doi.org/10.1002/spe.2525

[35] Jacob Krüger, Ivonne Schröter, Andy Kenner, Christopher Kruczek, and Thomas Leich. 2016. FeatureCoPP: Compositional Annotations. In *Proceedings of the 7th International Workshop on Feature-Oriented Software Development (FOSD 2016)*. ACM, New York, NY, USA, 74–84. https://doi.org/10.1145/3001867.3001876

[36] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. 2006. A quantitative analysis of aspects in the eCos kernel. *ACM SIGOPS Operating Systems Review* 40, 4 (2006), 191. https://doi.org/10.1145/1218063.1217954

[37] Miguel R Luaces, David Trillo Pérez, J Ignacio Lamas Fonte, and Ana Cerdeira-Pena. 2009. An Urban Planning Web Viewer Based on AJAX. In *Web Information Systems Engineering - WISE 2009 (Lecture {Notes} in {Computer} {Science})*, Gottfried Vossen, Darrell D E Long, and Jeffrey Xu Yu (Eds.). Springer Berlin Heidelberg, 443–453. http://link.springer.com/chapter/10.1007/978-3-642-04409-0{_}43

[38] Renaud Pawlak. 2005. Spoon: Annotation-driven Program Transformation — the AOP Case. In *Proceedings of the 1st Workshop on Aspect Oriented Middleware Development (AOMD '05)*. ACM, New York, NY, USA. https://doi.org/10.1145/1101560.1101566

[39] Ángeles S. Places, Nieves R. Brisaboa, Antonio Fariña, Miguel R. Luaces, José R. Paramá, and Miguel R. Penabad. 2007. The Galician virtual library. *Online Information Review* 31, 3 (jun 2007), 333–352. https://doi.org/10.1108/14684520710764104

[40] Christian Prehofer. 1997. Feature-oriented programming: A fresh look at objects. In *ECOOP'97 — Object-Oriented Programming*, Mehmet Akşit and Satoshi Matsuoka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 419–443.

[41] Ina Schaefer, Lorenzo Bettini, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-oriented Programming of Software Product Lines. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond (SPLC'10)*. Springer-Verlag, Berlin, Heidelberg, 77–91. http://dl.acm.org/citation.cfm?id=1885639.1885647

[42] Matthias Schöttle, Omar Alam, Jörg Kienzle, and Gunter Mussbacher. 2016. On the Modularization Provided by Concern-oriented Reuse. In *Companion Proceedings of the 15th International Conference on Modularity (MODULARITY Companion 2016)*. ACM, New York, NY, USA, 184–189. https://doi.org/10.1145/2892664.2892697

[43] Sandro Schulze, Jörg Liebig, Janet Siegmund, and Sven Apel. 2013. Does the Discipline of Preprocessor Annotations Matter?: A Controlled Experiment. In *Proceedings of the 12th International Conference on Generative Programming: Concepts and Experiences*. 65–74. https://doi.org/10.1145/2517208.2517215

[44] Nieraj Singh, Celina Gibbs, and Yvonne Coady. 2007. C-CLR: A Tool for Navigating Highly Configurable System Software. In *Proceedings of the 6th Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS '07)*. ACM, New York, NY, USA, Article 9. https://doi.org/10.1145/1233901.1233910

[45] Henry Spencer and Zoology Computer. 1992. # ifdef Considered Harmful , or Portability Experience With C News. *Usenix* (1992), 185–198.

[46] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban. 2005. AspectC++: an AOP Extension for C++. *Software Developer's Journal* 5, 68-76 (2005).

[47] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming* 79 (2014), 70 – 85. https://doi.org/10.1016/j.scico.2012.06.002 Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010).

[48] G. Vale, R. Abílio, A. Freire, and H. Costa. 2015. Criteria and Guidelines to Improve Software Maintainability in Software Product Lines. In *2015 12th International Conference on Information Technology - New Generations*. 427–432. https://doi.org/10.1109/ITNG.2015.75

[49] Eric Walkingshaw and Martin Erwig. 2012. A Calculus for Modeling and Implementing Variation. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering (GPCE '12)*. ACM, New York, NY, USA, 132–140. https://doi.org/10.1145/2371401.2371421

[50] Eric Walkingshaw and Klaus Ostermann. 2014. Projectional Editing of Variational Software. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences (GPCE 2014)*. ACM, New York, NY, USA, 29–38. https://doi.org/10.1145/2658761.2658766