Local features: Enhancing variability modeling in software product lines☆

David de Castro, Alejandro Cortiñas, Miguel R. Luaces, Oscar Pedreira *, Ángeles Saavedra Places

Universidade da Coruña, Centro de Investigación CITIC, Database Lab., Elviña, 15071, A Coruña, Spain

ARTICLE INFO

ABSTRACT

Context and motivation: Software Product Lines (SPL) enable the creation of software product families with shared core components using feature models to model variability. Choosing features from a feature model to generate a product may not be sufficient in certain situations because the application engineer may need to be able to decide on configuration time the system's elements to which a certain feature will be applied. Therefore, there is a need to select which features have to be included in the product but also to which of its elements they have to be applied.

Objective: We introduce *local features* that are selectively applied to specific parts of the system during product configuration.

Results: We formalize local features using multimodels to establish relationships between local features and other elements of the system models. The paper includes examples illustrating the motivation for local features, a formal definition, and a domain-specific language for specification and implementation. Finally, we present a case study in a real scenario that shows how the concept of local features allowed us to define the variability of a complex system. The examples and the application case show that the proposal achieves higher customization levels at the application engineering phase.

1. Introduction

Software Product Lines (SPL) support the semi-automatic development of families of software products. The products in the family are built from a set of common *core assets* and share many *features*, although they differ in others. The SPL development paradigm structures the development process in two phases: the *domain engineering* phase and the *application engineering* phase. The goal of the *domain engineering* phase is to analyze, design, and build the components of the product family and the SPL platform. A key activity of this phase is the analysis and modeling of variability, that is, the identification of the features that can be present in the products of the family and the relationships between them. The goal of the *application engineering* phase is the configuration and generation of a specific product. The application engineers select which features must be present in that product and the SPL platform generates it by adapting and combining the core assets according to the selected features (Weiss and Lai, 1999; Pohl et al., 2005). SPLs mean a great advance in the development of software product families since the automatic generation of these products led to a considerable reduction of development costs and an increase in product quality. This is the reason why there are more and more SPLs orientated to very different application domains (Weiss et al., 2006),

from the generation of software for airplanes (Sharp, 1998), software for IoT devices (Iglesias et al., 2019), web portals (Trujillo et al., 2007), or virtual stores (Rincon et al., 2015), just to name a few examples.

A *feature* represents a characteristic of a system and can be related to a functionality, a design decision, or other elements of the system (Pohl et al., 2005; Apel and Kästner, 2009b; Apel et al., 2016; Benavides et al., 2010). The variability of a SPL is usually described in a feature model, a hierarchical tree that represents the features that may or may not be present in any of the products of the family, and the relationships between them (Pohl et al., 2005; Apel et al., 2016). In the domain engineering phase, feature models allow domain engineers to define the variability of the SPL, which is essential in the design of the different components. In the application engineering phase, the SPL platform allows the application engineer to generate a product of the family by selecting the features that must be present in that product. In this way, it is possible to generate software that contains only those functionalities that are necessary. For example, although all e-commerce stores are very similar and share many features, there are some functionalities that may differ, such as the payment methods: some allow payment by credit card, some with PayPal, and some with
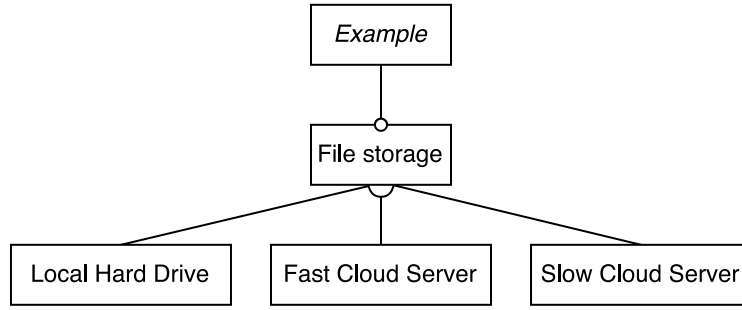

Fig. 1. Excerpt of the example feature model of a file storage system.
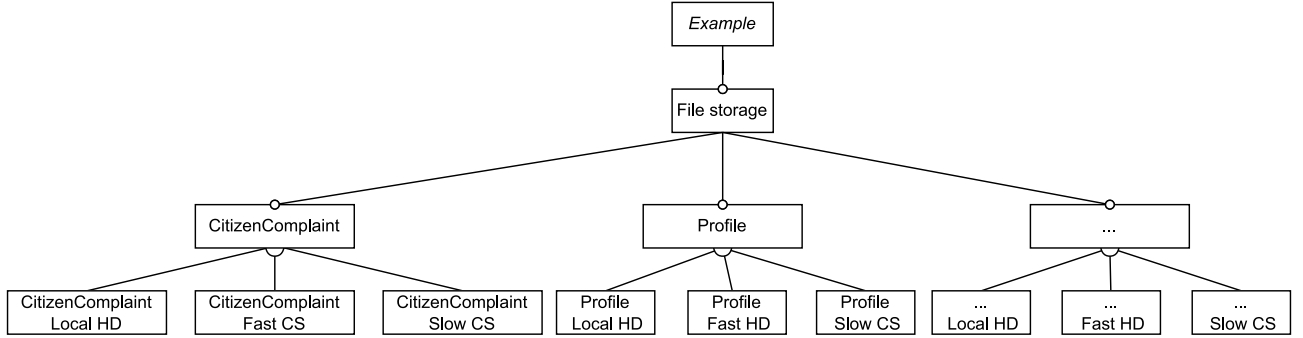

Fig. 2. Excerpt of the example feature model of a file storage system SPL supporting granular control.

both options. These payment methods are features of the product family and can be present in some products but not in others.

Managing variability with feature models is insufficient in many scenarios because the level of customization they allow can be limited for complex systems. This is why extensions to the feature model have been developed to improve variability modeling. Among these extensions are feature models with attributes (Czarnecki et al., 2002, 2005b; Benavides et al., 2005; Batory et al., 2006; Voelter and Visser, 2011), which allow adding additional information to a feature; or cardinality-based feature models (Riebisch et al., 2002; Czarnecki et al., 2002, 2004, 2005a), which allow defining cardinalities on subtrees of the feature model, that can then be replicated according to those cardinalities, therefore creating several instances of the same feature. Both extensions considerably improve the level of expressiveness of the feature models.

Problem Statement

In previous real projects where we applied the SPL paradigm, we encountered certain scenarios that highlighted the necessity for a more detailed level of product specification and customization beyond what traditional feature models could offer. The problem we have identified is that in certain situations, it is desirable to be able to express that some functionality of a product is applied only to specific parts of it, and these *parts* cannot be identified during the domain engineering phase but must be identified during the application engineering phase. Hence, we found that it is not sufficient to decide whether a feature is included or not in the product; we also need to specify to which elements of the system the feature will be applied. In other words, the features selected in the product configuration cease to be global (for the entire product) and become local (for specific parts of the product). Next, we present some examples to illustrate the motivation for this research problem.

Motivating examples

File storage. Many software systems must manage a large number of files. Consider, for example, a SPL for managing a family of document management systems for administrative processes in the public administration (with functionalities such as directory management, document storage, document review and approval workflows, etc.) The
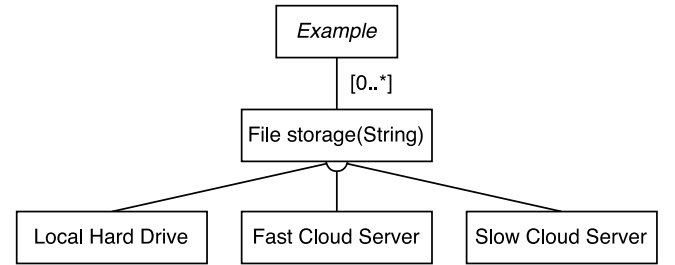

Fig. 3. Excerpt of the example cardinality-based feature model of a file storage system SPL.

data model of this software product line would have a large number of classes in which one of their properties would be a file (e.g., the employee record would have a profile photography, the regulation class would have the official gazette file, the citizen complaint class would have a collection of supporting evidences, etc.). This SPL could also have three alternatives for storing the files: storing them on a local hard drive, storing them on a cloud server with fast access, and storing them on a cloud server with cheaper but slower access time. Each of these alternatives would be represented as a feature in the feature model of the SPL (e.g., `Local Hard Drive`, `Fast Cloud Server`, and `Slow Cloud Server`, in the feature model excerpt shown in Fig. 1). Let us consider an additional requirement: since there is a substantial number of classes containing file attributes, potentially reaching the order of hundreds, which is justifiable in the context of an extensive document management software product family, we aim to have the capability to individually select the storage type for each file, providing granular control over the storage mechanism. The application engineer may need to select different storage options for different classes of files attending to the product's requirements and to optimize the access-time/cost trade-off. For example, in the case of a profile photograph, it is essential to serve it from a fast cloud server due to the significance of speed for optimal user experience. On the other hand, a gazette file,
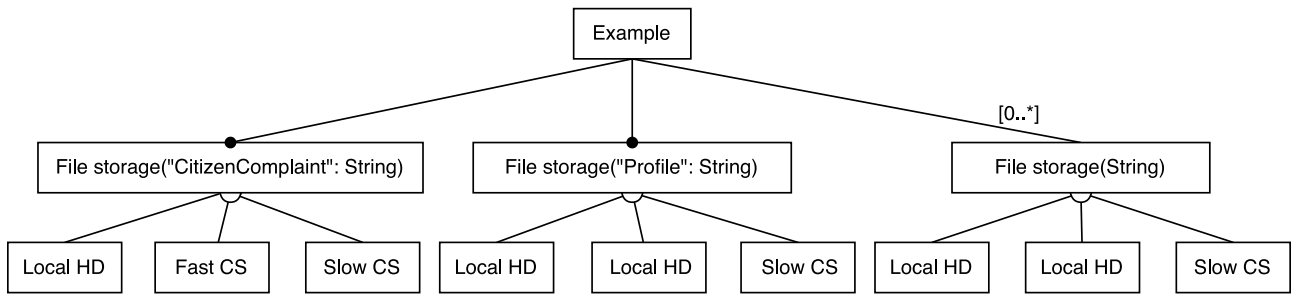
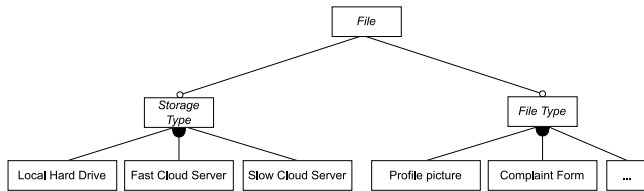**Fig. 4.** Specialization of the example in Fig. 3.

Fig. 5. Excerpt of the example solution with features binding for a file storage system SPL.

which may not require rapid access, can be served from the local hard drive. Lastly, citizen complaint evidences, being infrequently accessed, can be efficiently served from a slow cloud server.

There are already ways of addressing this scenario using the current state of the art:

(i) An approach would be to use a feature model similar to the one shown in Fig. 1. The application engineer would only be able to select one of the three features for a specific product, and it would be applied uniformly to all files across all classes. As a result, the SPL would not be able to meet the requirement that each file can utilize a different storage system.

(ii) The second solution would involve replicating the subtree shown in Fig. 1 as many times as the number of classes for which the `File Storage` feature can be applied. This would result in a feature diagram with dozens of replicated subtrees (see Fig. 2). During the application engineering phase, the engineer would have to indicate for each class which of the three storage alternatives would be applied. We believe that this solution leads to an unnecessarily complex feature diagram, which also necessitates a complex product configuration during the application engineering phase.

(iii) The third solution would involve applying cardinality-based feature models. The `File Storage` feature would be cloned as many times as there are classes with files in the data model (see Fig. 3, with the notation of Czarnecki et al. (2005a) for the cardinality-based feature model). Each clone of the subtree would be associated with a class of the model using an attribute in the `File Storage` feature (see Fig. 4). While managing the tree is simpler than in the previous case, during the application engineering phase engineers would still have to indicate the choice again for each class in the data model. Furthermore, the association between the features and the classes to which they will be applied is established as an attribute of the feature, which weakens such an association.

(iv) Another alternative is shown in Fig. 5. Our feature model could have a root feature `File` with two mandatory children, `Storage Type` and `File Type`. The feature `Storage Type` feature would have three children, each representing a file storage alternative (local HD, fast cloud server, and slow cloud server). The feature `File Type` would have as many children features as file

types we would need to store in the system (such as profile picture, for example). The decision of which storage option would be applied to each file type would be decided at the product's configuration time by binding child features of `File Type` with children features of `Storage Type`. Another possibility would be establishing the relationships between the file types and the applied storage types through cross-tree constraints in the feature model. The main drawback of this solution is that it would imply incorporating all the file types of a particular product into the feature model as features that may not be relevant to other products. This would generate an artificially complex feature model since it would have to incorporate elements that are not necessarily relevant features. In this sense, the feature model would become something like a supermodel that incorporates elements already defined in other system models, which we believe would be a bad design approach violating the principle of separation of concerns. In addition, we would still need to formalize how the binding between the two groups of features would be modeled or defined.

(v) Finally, another possible alternative would be having in our feature model an abstract feature `File Type` with as many children as types of files we would need to store in the system (again, examples include profile picture, gazette, etc.) These features could have an attribute `storageType` that would specify if that file is stored in a local hard drive, fast cloud server or slow cloud server (see Fig. 6). This alternative shares one significant drawback with alternative (iv), that is, we would be incorporating into the feature model features that represent the types of files of a particular product that may not be relevant for other products. In addition, the specification of the storage type of the files would be expressed in a weak way, since these storage alternatives would no longer appear as features in the feature model (and we consider they are features that should appear in the feature model).

We believe that these five solutions can be improved. In the case of option *(i)*, the solution falls short of meeting the requirements during the application generation phase, as it does not allow the engineer to configure which alternative would be applied to each class. Regarding solutions *(ii)* and *(iii)*, we find them to be complex, as the feature model repetitively includes the same subtree numerous times, which could be avoided. Additionally, the engineer would need to indicate the chosen storage alternative for each class, even if all classes use the same alternative and only one of them uses a different one. In the case of solution *(iv)*, the feature model would have to incorporate elements already defined in the data model as features, resulting in an artificially complex feature model. Finally, in the case of solution *(v)*, the features representing the variability regarding storage options would disappear from the feature model (something we consider a bad design), and the relationship between the file types and the applied storage option would be defined in a weak way.

Scenarios similar to the one presented in the previous example can emerge when features represent transversal elements of the software, as exemplified in the following two examples.
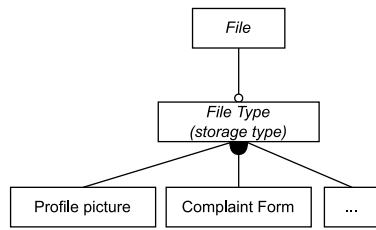
Fig. 6. Excerpt of the example solution with features representing file types with attributes for defining the storage alternative.

Access Logging. A common requirement for many software systems is to maintain a log of who (and when) performed modifications on the data (in some cases, it is also necessary to keep the original and modified data). This requirement can be included as a functionality in the feature model. This way, the application engineer would have the possibility to specify for each generated product whether the functionality is included or not. However, in most cases, the application engineer would not want this functionality to apply to all data in the system but only to those for which user access auditing may be required. If the application engineer were to apply the functionality to all data, it would entail storing and managing logs that the application engineer does not actually need, which we consider a poor design decision. Therefore, in this example, the application engineer would desire the ability to include the functionality in the product but also to specify which data in the system it will be applied to. The situation is similar to the previous case. The application engineer wishes to select a functionality but does not want it to be applied to all elements in the data model; instead, they want to choose specific ones.

Data Export. Many enterprise applications allow users to export data in formats like CSV for certain classes in the data model. Data export in CSV can be modeled as a feature in a feature model. However, similar to the previous examples, we may not want this functionality to be available for all classes in the system's data model, but only for those that are required based on the specific software product's requirements.

Definition: Global and Local Features

We faced scenarios similar to these ones in real projects, which led us to propose the concepts of *global features* and *local features*. Our proposal includes defining software variability at two levels: *global*, which comprises features that apply to variation points defined in the domain engineering phase and *local*, which comprises features that apply to variation points defined in the application engineering phase. A global feature retains the semantics found in current feature models; it represents features that can be present in any of the products within the product family. On the other hand, a local feature in a software product line is a functionality that, when selected for inclusion in a product, will be applied solely to specific elements defined in other system models. That is, during the domain engineering phase, the engineer determines that some features may be applied to certain elements of the system models. During the application engineering phase, the engineer must associate these features with the specific elements of the system (specified in other system models) to which they will be applied. Our proposal includes using multimodels to specify the binding (Siegmund et al., 2020) between local features and the elements specified in other system's models. Thus, the formality of the proposal is improved with respect to other alternatives such as using a configuration file at the application engineering phase.

Fig. 7 shows how the previous example would be solved with the concept of local features. In the domain engineering phase, the feature model would specify that there are three alternatives to store files. However, these features are not associated with predefined variation points of the SPL, since, in the excerpt shown in Fig. 7, the three features `Local Hard Drive`, `Fast Cloud Server` and `Slow Cloud Server` are all local features. In the application engineering phase,

the engineer can select a global feature to be applied to all files in all classes by default if no local features are specified. Furthermore, the engineer may select local features to be applied to specific elements of the data model. In the example shown in Fig. 7, an excerpt from the data model has two classes, `Profile` and `CitizenComplaint`. The class `Profile` is associated with the feature `Fast Cloud Server`, as it may be necessary to retrieve these files quickly. Similarly, the class `CitizenComplaint` is associated with the feature `Slow Cloud Server`, since evidence files require a lot of storage space and are not often accessed. The files in other classes (e.g., the regulation class) are stored as local files because `Local Hard Drive` was selected as the global feature. In this way, we could select a predefined storage alternative for all classes, but we would be able to customize the storage alternative for specific classes by associating them with local features. This approach leads to simpler feature models and simpler product configurations in the application engineering phase.

In the rest of the article, we develop the concepts of global and local features and how they can be implemented in practice. From the point of view of modeling and specification, we decided to implement global and local features through the concept of multimodels, which has already been used in some previous works as a way to specify the relationships between feature models and other system's models (Gonzalez-Huerta et al., 2013). From the point of view of the practical specification of global and local features, we propose using a Domain-Specific Language (DSL) that enables the application engineer to define a product specification including the aforementioned relationships between local features and other system elements. We introduced the concept of local features informally in a previous conference paper (de Castro et al., 2022), in which we showed a specific industrial experience that motivated the need for local features in some SPLs.

Finally, the article presents an application case in a real scenario in which we applied the concept of local features to a SPL for the generation of geographic information systems (GIS), that is, systems in which many of the entities have a geo-spatial component as part of their attributes. This application case shows how the concept of local features allowed us to specify and configure how the features related to data visualization can be applied and adapted to the user visualization requirements in a real system.

The rest of the paper is organized as follows. Section 2 provides an overview of previous work. Section 3 explains our proposal, conceptually, with a brief example, and shows the changes that had to be made to our SPL's variability models to support it. Section 4 presents a case study to illustrate our proposal with an existing GIS web application, showing an example step by step. Finally, Section 5 concludes the paper and comments on some ideas for future work.

2. Background and related work

2.1. Software product lines and feature modeling

Software Product Lines (SPL) are one of the solutions adopted by the software industry to develop quality software in reduced time. An SPL is a platform that supports the development of a family of software products that share a set of common *features* but that can vary in others. To semi-automate the development of one of the software products of the family, an SPL allows the engineer to select the features that should be present in that product. Then, the SPL assembles and adapts a set of *core components* to generate that product based on the selection of features. The *variability* in the features of the product family is represented in a *feature model*, which represents the features of the product family and the relationships between them. This approach tries to bring to software development the production schemes that have been applied in the last century in other industries, such as the production of automobiles, for example. There are many examples of applications of SPL in real scenarios. For example, companies in the
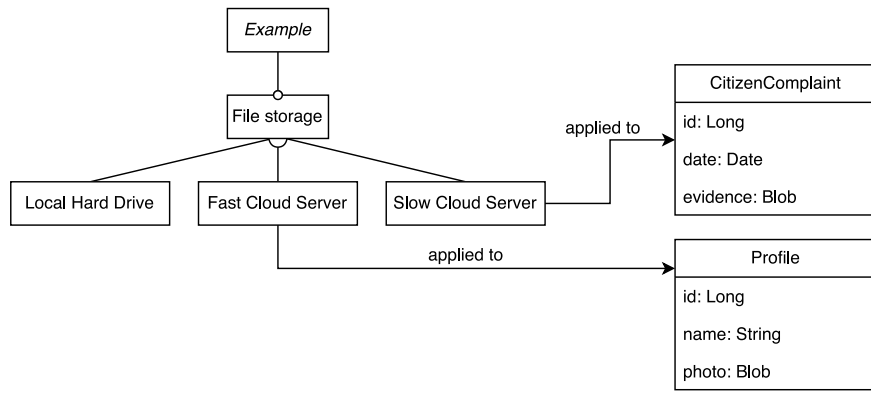
Fig. 7. Example of application of local features to a public administration system.

aviation sector have SPL oriented toward generating software for their aircrafts (Sharp, 1998), and there are SPLs oriented toward generating e-commerce web applications (Rincon et al., 2015).

Feature modeling (Kang et al., 1990) is the *de facto* variability representation for SPLs (Sousa et al., 2016; Benavides et al., 2010; Galindo and Benavides, 2020; Apel and Kästner, 2009a). A feature model is a tree where the features (*end user-visible characteristics of a software system* (Kang et al., 1990)) of a product line are hierarchically structured. Each feature can be decomposed into several sub-features, and they can be mandatory, optional, or alternative features (Pohl et al., 2005). Every product in the product line is specified by the set of features included in it. Besides the relationship between a feature and its sub-features, we can define cross-tree constraints between the features, for example, `including feature A implies that feature B is also included`. These are the components of what can be considered basic feature models. However, previous works have proposed several extensions to these basic feature models (Benavides et al., 2010; Alférez et al., 2019).

Cardinality-based feature models (Riebisch et al., 2002; Czarnecki et al., 2002, 2004, 2005a) allow defining UML-like multiplicities or cardinalities for the features. These cardinalities determine the number of instances of a feature that can be included in a product, and each of these features can include a specific set of sub-features. Extended feature models (Czarnecki et al., 2002, 2005b; Benavides et al., 2005; Batory et al., 2006; Voelter and Visser, 2011) allow to link features with *attributes*; this is, each product can include extra information for a selected feature beyond its own selection. These attributes can be, for example, a number within a specific range, or a string literal, and can be used within the constraints between the features. Both of the approaches have been used together (Czarnecki et al., 2002, 2005b; Karataş et al., 2013) to obtain a flexible and complete variability model.

However, these extensions to basic feature models only allow for expressing certain variability elements in an SPL. Specific domains require additional models and relationships between them to express additional variability.

2.2. Multimodel description of software product lines

A *multimodel* is a set of interrelated models, each of them representing a different viewpoint of the system. Each *viewpoint* is an abstraction of a system part with a specific purpose and is represented through a specific model (Barkmeyer et al., 2003). In a multimodel description, we can establish relationships between these different viewpoints (models) so that one element of one viewpoint can be related to an element of another. The relationships between elements of different models are defined outside those models, in a specific model of relationships.

Fig. 8 shows a metamodel of multimodels (González-Huerta et al., 2012). The left side of the figure contains the classes representing a

model of the system. As we can see in the figure, the class `ViewpointModel` represents a particular model, which is composed of `Entities` and `Relationships` between those entities. There will be as many viewpoint models as there are of interest in the domain, and each viewpoint model will, in turn, have as many entities and relationships as necessary to represent the concepts and relationships in the corresponding view model. On the right side of the figure, the rest of the classes represent the concept of multimodel and its components. A `Multimodel` is composed of `ViewpointModels` and `MultimodelRelationships`. A `MultimodelRelationship` defines the relationship between `MultimodelEntities`, each of them extending an `Entity` of a specific `ViewpointModel`. In addition, the multimodel can also contain `MultimodelAttributes` associated to either a `MultimodelEntity` or a `MultimodelRelationship`.

Previous works have already used the concept of multimodel to represent information of a software product line. González-Huerta, Insfram, and Abrahão (Gonzalez-Huerta et al., 2013) used multimodels to model the relationships between features, elements of the software architecture, and quality attributes of the software product line. This multimodel allows the platform to assess the fulfillment of non-functional quality attributes based on the selection of features and the presence of specific software architecture components in the definition of the product to be generated. The information contained in the multimodel even allows to make recommendations on the features that should be selected to meet those non-functional requirements.

We have decided to use the multimodel approach instead of CVL or OVM (Pohl et al., 2005; Metzger et al., 2007) because we found it to be a straightforward approach to establish relationships between features and other elements. Additionally, it has been used in previous works to establish similar associations, and in terms of practical implementation, it seemed like an easy and manageable option. Moreover, both OVM and CVL can be defined using multimodels, so we are using a more general approach. On the other hand, CVL was dropped due to legal, patent-related issues (Berger and Collet, 2019). Similarly, OVM aims to separate the description of feature variability from the rest of the software artifacts providing a powerful way to handle feature variability and configuration, but it is not specifically designed to represent relationships between models in a direct or formalized manner. Also, in OVM the association between the features and the variation points is defined in the domain engineering phase, and it is not expected to be modified in the application engineering phase.

3. Local features in software product lines

In this section, we introduce and formalize the concept of global and local features. We first present the formalization of global and local features using multimodels, and then we provide an example of its application.
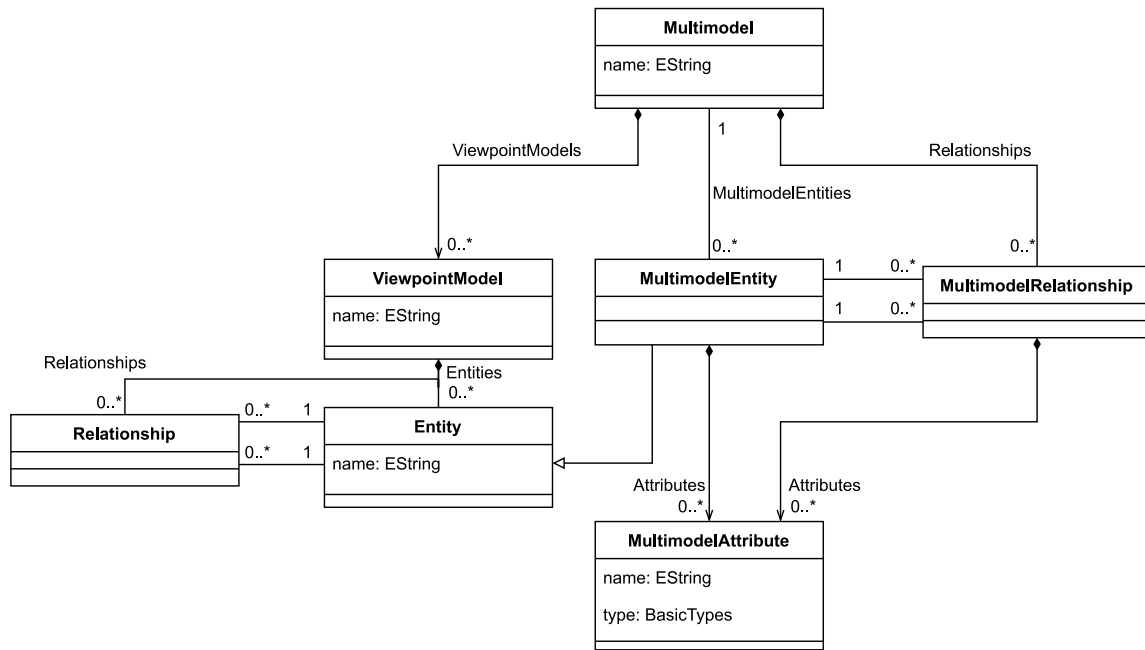
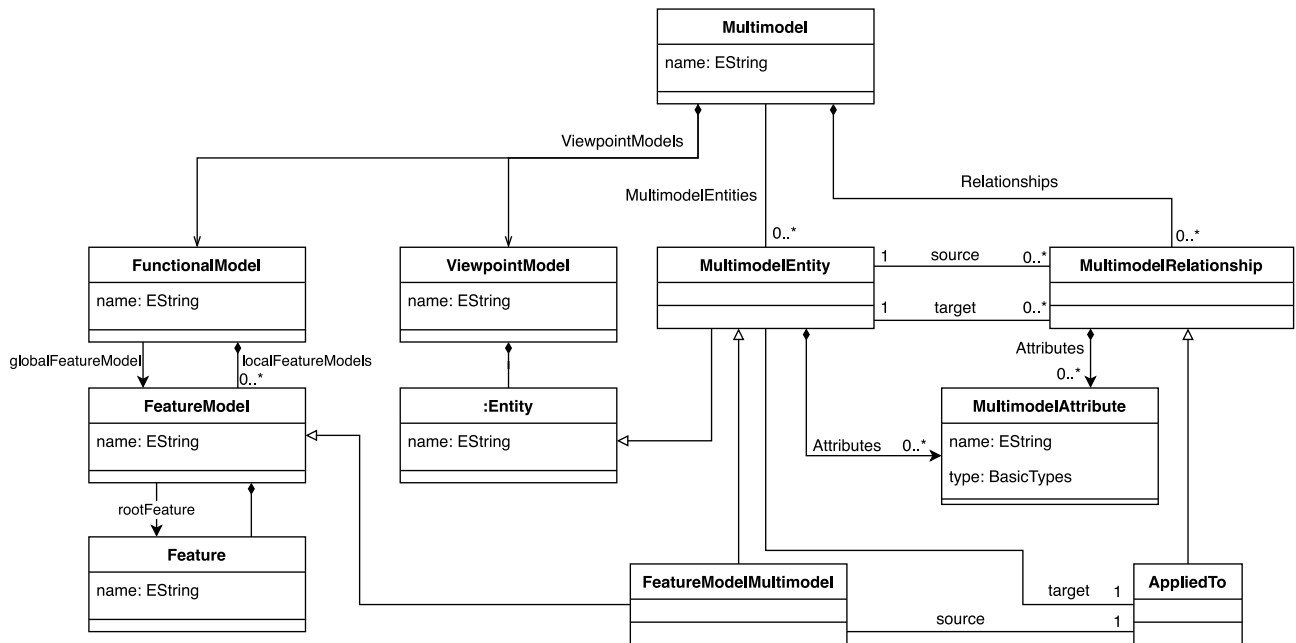**Fig. 8.** A metamodel of multimodels (González-Huerta et al., 2012).

Fig. 9. A metamodel of multimodels with the inclusion of local features.

3.1. Definition of global and local features

The definition of global and local features assumes that the products of the family supported by the SPL are defined by multiple models. One of these models must necessarily be a feature model. A *local feature* is a feature in a software product line that, if selected to be included in a product, it will be applied only to specific elements defined in different system models, as defined by the engineer. A *global feature* has the semantics they have in current feature models. It represents a feature that can be present in any of the family's products. As we have already mentioned, local features will be applied only to specific elements of the system defined in other system's models. The other system's models can represent different viewpoints of the system, such

as its architecture, data model, or visualization configuration. The relationship between a local feature and the elements in other models to which it will be applied is modeled using multimodels. As we can see in Fig. 9, our multimodel for a SPL comprises all the models of the system (i.e. a `FunctionalModel` and other `ViewpointModels`) and defines a `MultimodelRelationship`, `AppliedTo`, that associates a local feature of the feature model with any `MultiModelEntity` in another model. Notice that a `MultiModelEntity` can represent different things in different metamodels. For example, a class can be a `MultiModelEntity` in a certain viewpoint of the system, but a method can also be a `MultiModelEntity` in a different viewpoint model. Notice also that, since `AppliedTo` inherits from `MultiModelRelationship`, it can only connect sources and targets

System's Model
(Product configuration)

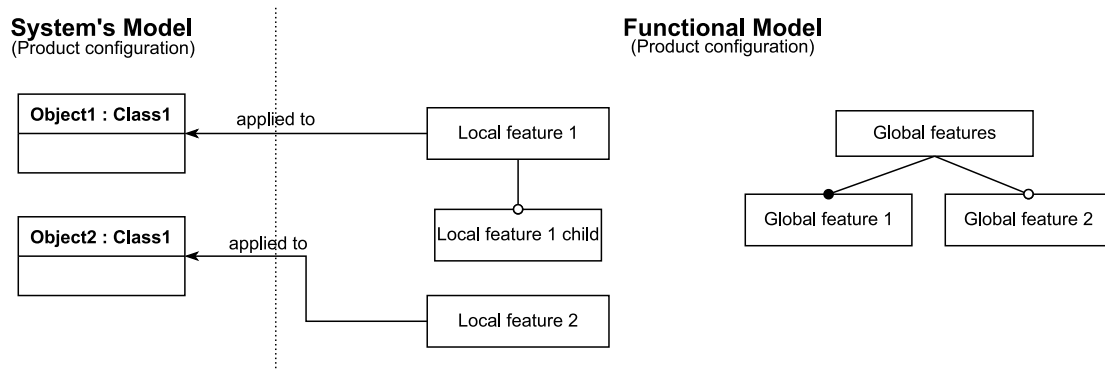**Functional Model**
(Product configuration)

Fig. 10. Definition of local features in SPL.

which are both `MultiModelEntities`. Thus, we incorporated to the multimodel the class `FeatureModelMultimodel`, which just inherits from `FeatureModel`. In Fig. 9, it can also be seen that a `FunctionalModel` is composed of many `FeatureModel`. One of them is the global feature model, and the other ones are the local feature models.

Throughout the article, we will explain with examples the existing relationships between different models with the feature model for the application of local features. To illustrate these relationships, we will use columns to represent different models of the system, and we will use arrows with the annotation *applied to* to indicate the association of features with elements of the other models. Fig. 10 shows a simple example. The figure illustrates the product configuration of an unspecified *system model* with two objects of the same class on the left column. On the right column, it depicts the functional model consisting of two local features that apply only to those objects (*Local feature 1* and *Local feature 2*) and a global feature model with features that affect the complete system (*Global features*). In this example, you can observe that for *Object1*, only *Local feature 1* is applied, whereas for *Object2*, *Local feature 2* is applied. It is important to note that due to space constraints, the tree of local features has not been depicted alongside the tree of global features. However, these local features would also be present within the global feature tree to enable the definition of global behaviors. Additionally, this setup allows the overwriting of specific behaviors only for the objects of interest. The behavior of these local features will be discussed in more detail below with concrete examples.

3.2. Example

In order to see how local features are applied in practice, the following is a case example of an SPL oriented to the generation of e-commerce stores. The applications generated by the SPL are intended for the sale of products that can be very varied: from digital products, such as films, songs, books, etc., to physical products, such as pencils or pens. These products can be found sorted by categories and even sub-categories. Payment can be made online and, finally, these applications will have a product catalog where products are grouped by categories.

Fig. 11 shows the multimodel of this SPL, which consists of two models: a catalog model (on the left) and a set of feature models (on the right). The catalog model consists of a class `CategoryComposite` that inherits from `Category`, which has only one attribute named `name`. These two classes represent the categories and subcategories of the products using the composite pattern. Hence, the application engineer can describe the catalog model of an e-commerce store by defining a tree of categories.

The right side of Fig. 11 shows the functional model, which contains a global feature model (on the right) and a local feature model (on the left):

- **E-Commerce**: this is the global feature model, i.e. the selection of one of the features in this feature tree affects the whole product. Within this tree are the features related to payment methods (`Payment`), whose payment can be made by bank transfer (`BankTransfer`) or by credit card (`CreditCard`). In turn, there is a second feature, `Catalog`, which adds the functionality of being able to visualize the products in catalogs grouped by categories. The features within its feature tree are grouped in two subtrees: `Preview` and `Layout`. The `Preview` features subtree contains the features related to the preview of the products in the catalog, so depending on the feature selected here the product preview will change. The `NoPreview` feature does not activate the product preview, `AudioSnippet`, `VideoSnippet` and `TextSnippet` activate the audio, video or text preview, respectively, adding the corresponding audio, video or epub player. The other sub-tree of features, `Layout`, contains the features associated with how the different products are arranged within a catalog; this can be in the form of a grid (by selecting the `Grid` feature) or in the form of a list (`List` feature).
- **CategoryDisplay**: this is a local feature model that is a subset of the `CategoryDisplay` tree of the global feature model seen before, so that each `Category` class of the data model is related to a `CategoryDisplay`. Hence, for each category that is created by the application engineer, could have a `CategoryDisplay` feature tree that defines the specific preview and layout of the products of the category. In the case that this does not exist, the behavior that this `CategoryDisplay` would be the default one defined in the `E-Commerce` tree.

In Fig. 12, we show an instance of the model in Fig. 11 representing the configuration of a specific product. On the left side on the figure, there is the specification of the different categories of the product. On the right side, the chosen functionalities (from the feature models) that the product will have (both global and local ones that apply only to certain categories). For this example we have defined two main categories, *OfficeSupplies* and *DigitalContent* (instances of the CategoryComposite class) and two subcategories, *Pencils* and *Films* (instances of the Category class). The category *Pencils* is not associated with any local feature. Being a physical product, the more suitable way to represent them in a store is through a list and without preview. Since the global feature model already has the default behavior of listing the elements of a category (`List`) and not having a preview (`NoPreview`), no local feature has been associated to it to change this behavior. On the other hand, there is the *Films* category, which, unlike the previous category, can be previewed. Hence, it is associated to the feature `VideoSnippet` through the applied local feature model with the root feature `CategoryDisplay`. The feature `Grid` is also selected for this category, since this type of layout can be more attractive to potential buyers and gives prominence to the preview. Finally, there are other global features that affect the entire product; in this
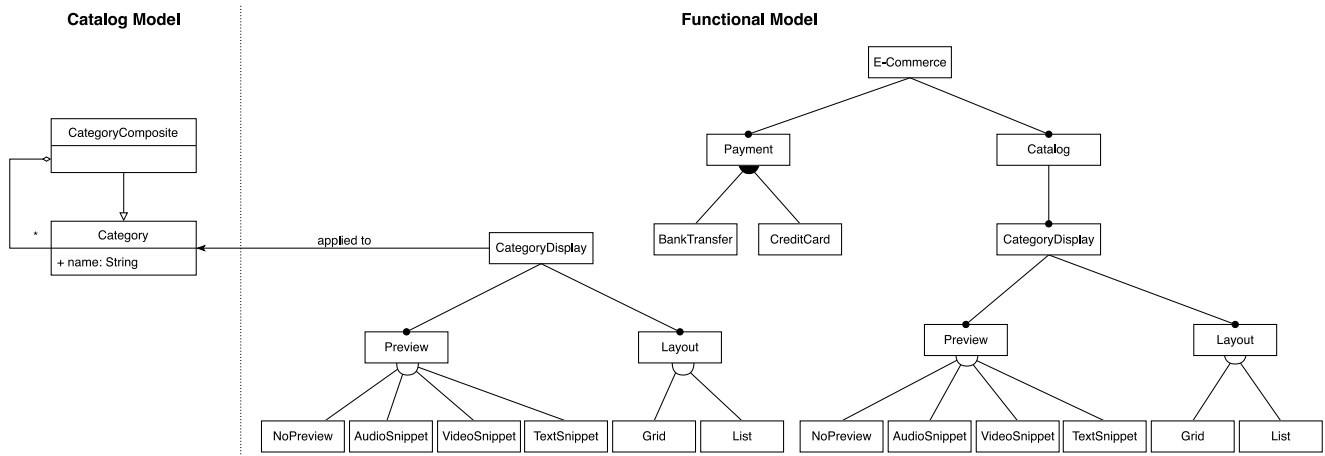
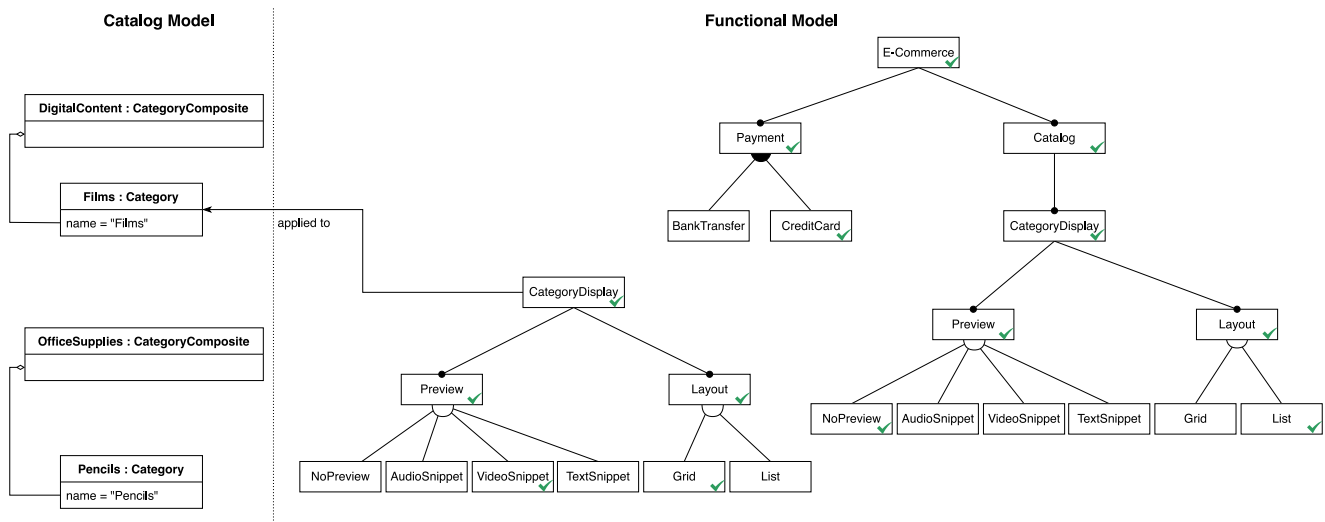**Fig. 11.** Sample SPL model for the generation of e-commerce applications.

Fig. 12. Example of product configuration with the SPL.

case, the feature `Payment` and its selected subfeature, `CreditCard` (so you can only pay by card), and the mandatory feature `Catalog` (the selection of these characteristics is represented in the diagram with green checks to facilitate comprehension).

By modeling variability in this way, making use of local features, it is possible to have different ways of previewing according to the nature of the product that is for sale. If the model did not have local features, the generated product could not have the preview of the products since pencils are not a video and therefore cannot contain a preview. However, by having local features, it is possible to configure the product as shown in Fig. 12. This example shows the practical usefulness of local features and how they improve the level of expressiveness of the feature models. Not only can we choose the features we want our product to have, but we can also specify which specific elements of the product to be generated will have those features.

4. Case study: developing geographic information systems with local features

In this section, we describe the case study that in fact motivated the need for local features. First, we introduce the context for the case study, and afterward, we describe how our proposal was applied and implemented in our case.

4.1. Context

In a previous work, we addressed the development of Geographic Information Systems (GIS) with SPLs (Cortiñas et al., 2017). The main characteristic of GIS is that they manage entities with a geospatial component. This characteristic affects all the software layers, from the database, which has to support geospatial data types and operations, to the user interface, which usually presents the information in maps and layers. GIS also support specific functionalities, such as route calculation or data processing based on their spatial representation. These systems are intensively used in public administrations and private companies since they are a typical tool in managing infrastructures (such as supply or transportation networks) or mobility scenarios (such as logistics or mobile workforces), for example.

Organizations such as ISO[1] or OGC[2] have made an important standardization effort in GIS. The result is that most GIS have the same software architecture and share tools, libraries, and software components in their development, independently of their application domain. Furthermore, GIS in different application domains share many functionalities. This was the motivation to explore the application of an SPL approach to GIS: our SPL comprises the features that may appear in a

---

[1] International Organization for Standardization, https://iso.org
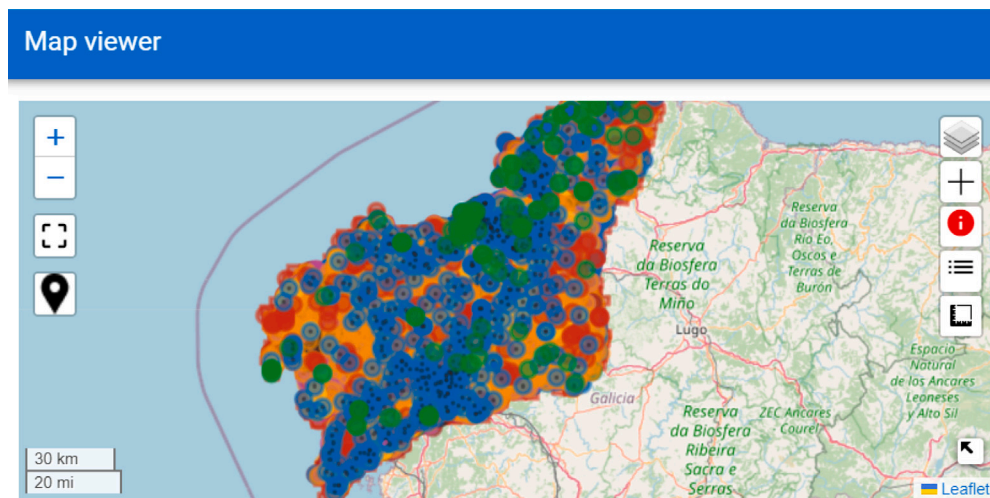[2] Open Geospatial Consortium, https://www.ogc.org

Fig. 13. Example of a product generated with a single map viewer.

GIS and allows the engineer to select which of them must be included in a specific product.

At the Databases Laboratory[3], we have been working on a Software Product Line that generates web-based GIS applications; that is, web applications that allow users to visualize and interact with geographic data, mainly through maps, as well as to offer other functionalities that were identified as common for GIS after an analysis of the domain (Cortiñas et al., 2017). This SPL has been employed in the industry for developing several products, including heritage management, facility management, reduced-mobility accessibility, and public transport management (see Fig. 13). Some of these products are large-scale, such as WebEIEL[4]. This is a product family with similar functionalities such as map viewing and user location. However, managing the functional variability of different products is not enough in the domain of GIS. The main difference when developing two GIS appears in the model of the domain. For example, a GIS product for a parcel company requires the system to manage the drivers, warehouses, and roads. In contrast, a GIS oriented to promote tourism in a region requires managing hotels, attractions, or natural landscapes. The data model or domain of a GIS affects its whole functionality, starting with the information that is shown in the map viewers and the way this information is drawn. Therefore, the architecture of our SPL is described by three models: a feature model, a data model, and a visualization model. The variability of functionalities is modeled as a feature model, through which the domain engineer can select which of them are required for each of the GIS products to generate. The data model is defined as an UML class diagram that describes the entities, properties and relationships between entities for the product. Finally, the visualization model allows configuring the different maps that the application will have, as well as the layers and styles with which they will be represented.

During these projects, it was observed that while time-to-market was significantly reduced, the resulting products always presented a common issue. The functionality defined through feature selection in the feature model is consistently applied in a general manner, without the ability to customize it for specific parts of the system. For example, if a functionality affecting the map viewer of the application was selected, it would be applied in the same way across all maps in the generated product. However, some features may need to be applied only to certain elements of the system. For instance, a GIS generated with the SPL would have different maps, and certain features would be necessary for some maps but not for others. Current feature modeling options do not account for this, requiring additional adaptation efforts for each product that software developers must carry out, resulting in a separation of the product from the product family. Therefore, it is necessary to be able to specify features that affect only a specific part of a system.

WebEIEL[5] is a web application dedicated to publishing geographic information collected by the Provincial Council of A Coruña, Spain, that we will use a paradigmatic example. Most of the data handled by WebEIEL has a geographic dimension, and it is shown to the user via map viewers with different layers (each layer represents one entity), styled depending on the data represented. There are different types of map viewers. For example, sometimes the application shows a full-fledged map viewer that displays a large collection of layers and provides complex functionality to the user; and sometimes the application requires just a simple map viewer that displays one or few layers and provides almost no functionality to the user beyond its visualization.

WebEIEL's data model is composed of entities that describe infrastructures (e.g., road network) and facilities (e.g., hotels, parks, etc.) related to the municipality where they are located. The data model is not very complex since most of the relationships are spatial (they do not require foreign keys in the database) and, at the same time, it is extensive because it includes around a hundred entities. Among the entities, there are different types regarding the functionalities of the application. For example, there are entities that should not be modified, such as the *municipalities*, whereas other entities need to be created and edited from the application, such as *water treatment plants* or *hotels*.

The first version of WebEIEL was developed in 2008–2010, and was operational until December 2022. It has been technologically outdated for a long time, as resources to its maintenance were scarce. In 2022, we started a project to upgrade WebEIEL to use current technology, using a previously implemented Software Product Line for the generation of web-based Geographic Information Systems (Cortiñas et al., 2017), evolved in a posterior work which describes the DSL used to generate products of this SPL (Alvarado et al., 2020). The version of WebEIEL currently deployed has been generated by the SPL, and modified afterwards by a development team. An example of a map viewer, a concept used all over the section, can be seen in Fig. 14, and accessed in the actual application[6].

---

[3] Databases Laboratory website: https://lbd.udc.es/.
[4] https://webeiel.dacoruna.gal

[5] WebEIEL website: https://webeiel.dacoruna.gal
[6] Example of map viewer showing hotels, among other layers, in WebEIEL website: https://webeiel.dacoruna.gal/map-viewer/movilidad?hash=1677844039765

Fig. 14. WebEIEL screenshot.
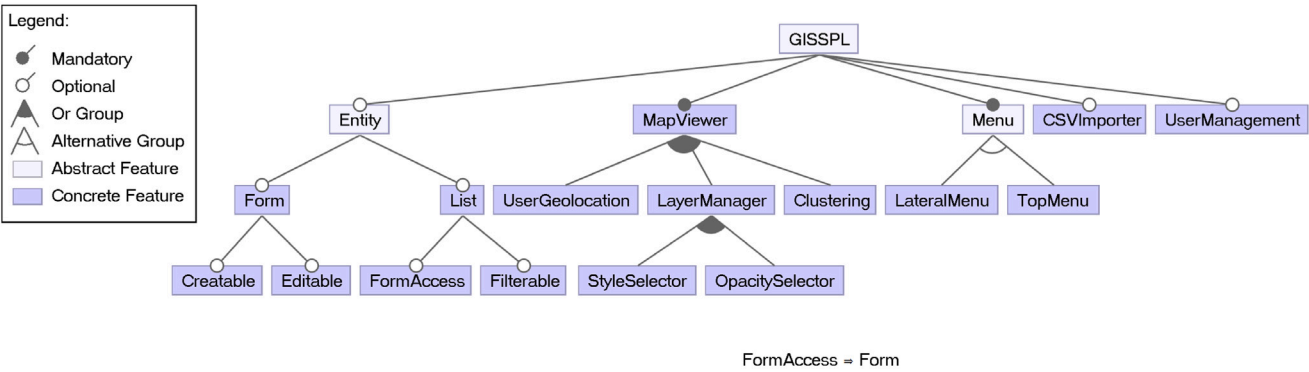

FormAccess ⇒ Form

Fig. 15. Product Line Feature Model (simplified excerpt, only 18 of the 175 features of our FM are displayed).

Briefly explained, our SPL generates web-based GIS that can be specified using three models: a feature model (see Fig. 15), used to determine the functionalities of a product; a data model (see Fig. 16), that allows the application engineer to define the entities and relationships of the domain of a specific product; and a visualization model (see Fig. 17), so the application engineer can configure a product to have different map viewers, each one showing different layers with data. By creating multiple maps, the application engineer can categorize information and tailor the maps' content to specific user groups, making them more appealing and user-oriented. For instance, a map designed for tourists might display only hotels and parks, while another map tailored for a city council worker could show the running water networks. Moreover, this approach significantly enhances the application's performance by avoiding the slowdown caused by loading too many layers simultaneously.

In the feature model (see Fig. 15), the left-most branch defines features related to the entities defined in the data model (`Entity`). `Form` generates detailed views of the entities. Depending on the sub-features `Creatable` and `Editable`, these detailed views provide the possibility of creating new elements, or editing existing ones. The feature `List` generates listings for each entity, in which all of its elements are displayed in a paginated table. These listings may allow the user to access the detail view of the elements (feature `FormAccess`, which implies the feature `Form`), and may be filterable (feature `Filterable`).

The second branch defines features associated with the maps (`MapViewer`). The sub-features shown activate functionalities such as geolocating the user (`UserGeolocation`), grouping the geographic elements of a layer by their location (`Clustering`), and managing the map layers (`LayerManager`). The latter allows the user to hide and
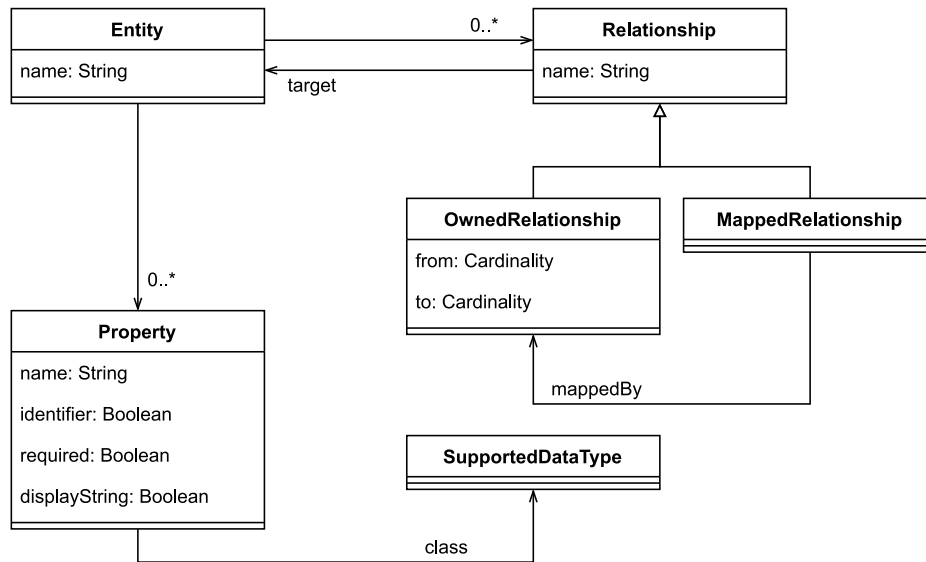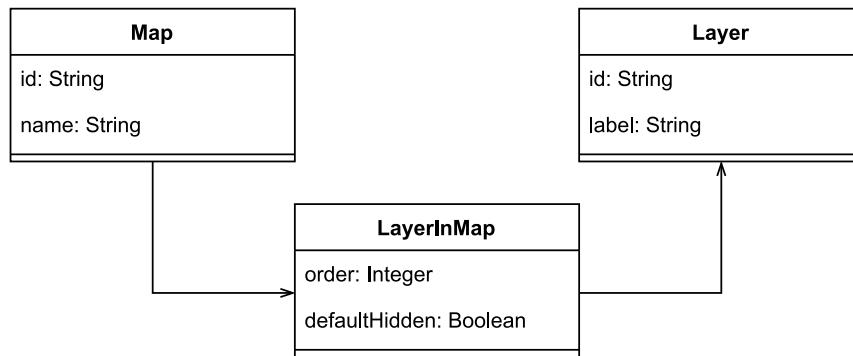
Fig. 16. Product line data metamodel.

Fig. 17. Product line visualization metamodel (simplified excerpt).

show a specific layer of the map, and also includes two sub-features: one to switch the style of a layer (`StyleSelector`), and other to change the opacity of a layer (`OpacitySelector`).

The last branches define features to decide if the application has a horizontal menu in the top or a left sided vertical menu (`Menu` and its children); to include a CSV importer that allows loading data into the database from a CSV file (`CSVImporter`); and to handle user registration and authentication (`UserManagement`).

The data model (see Fig. 16) is one of the three models of our multimodel. It allows the application engineer to specify the domain of a product, and is somehow a simplified version of the *Object Relational Mapping* model, also very similar to UML. The application engineer defines the entities represented by the `Entity` class, the properties of the entities in the `Property` class and the relationships between the entities represented with the `Relationship` class. This difference with respect to ORM/UML is that our model does not provide support for defining everything that is possible to do with ORM/UML, such as inheritance.

Finally, we have the visualization model (see Fig. 17) where the user can define maps (`Map`), layers that present different data (`Layer`), and the relationship between these two elements, that serves to identify which layers belong to each map (`LayerInMap`). The represented model is a condensed version of the complete model, aimed at simplifying the solution's comprehension. To achieve this simplification, all classes related to styles have been omitted. These classes are responsible for defining how a layer is visualized in a map, including aspects such as color representation and geometry opacity.
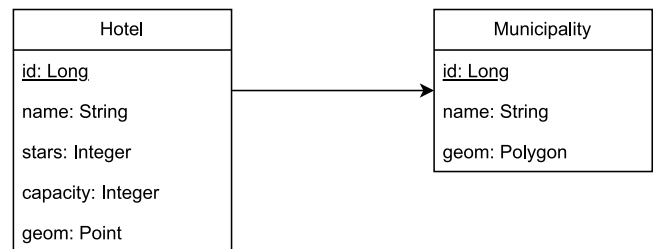

Fig. 18. Example of a simple data model for a web-based GIS.

Going back to WebEIEL, we have selected, for our example, simplified excerpts of the data model, shown in Fig. 18, and of the visualization model, shown in Fig. 19. The data model contains two entities, *Municipality* and *Hotel*, whereas the visualization model contains two maps: *MunicipalityMap*, that contains a layer to represent municipalities (*MunicipalityLayer*), and *HotelsMap* which contains a layer that represents hotels (*HotelsLayer*), and a layer that represents municipalities (*MunicipalityLayer*). Both maps also contain a base layer (*BaseLayer*) that will work as map background.

We have already mentioned above that the requirements for the entity *Municipality* are different from the ones for the entity *Hotel*: we do not want the users of our product to create, edit nor remove *municipalities*, but they can do that for *hotels*. Therefore, since at least one of the entities for the product require these functionalities, we need
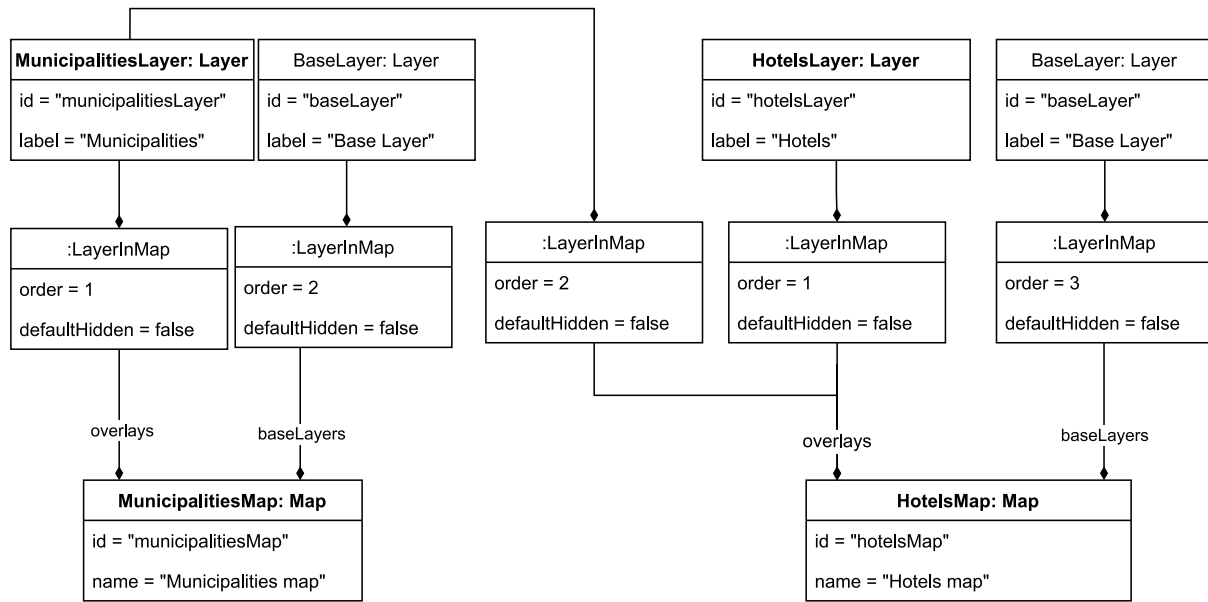
Fig. 19. Example of a simple visualization model for a web-based GIS.

to select the features `Creatable` and `Editable` (see Fig. 15). The problem is that these functionalities will be affecting all the entities of the data model, and then a user would be able to modify a *municipality*. To prevent this, a developer needs to modify the generated source code of the product and *remove* functionalities for some of the entities.

Let us assume that the *MunicipalityMap*, since it only has one overlay, do not require a *layer manager*, whereas the *HotelsMap*, having two overlays, requires a *layer manager*. This is the same case than before: since at least one of the map viewers needs to have a *layer manager*, the analyst has to select the feature `LayerManager`, and a developer needs to remove the functionality after the product is generated.

Obviously, having to modify the generated products straight after the generation only to remove features is far from ideal, since it rises the time to market, the maintenance costs, and the evolution complexity for the whole product family of the product. In the next section we explain how we adopted *local features* to solve the problems described, showing the changes made to the design of the product line, and also how we approached the implementation.

4.2. Developed solution

First step is to identify *local features*. We can do that by splitting the original feature model (see Fig. 15) into pieces. The result is shown in Fig. 20, where we can see four different feature models. One of them, GIS-SPL, is a global features model and it is not linked to any element, while the other three root features can be *applied to* specific elements of the other models, so they are local feature models (we omitted all the elements which have no local features applied). Thus, the `EntityFeature` is applied to the `Entity` class of the data model, whereas the `MapFeature` and the `LayerFeature` are applied to the `Map` the `LayerInMap` classes of the visualization model, respectively. Hence, when instantiating each of the linked elements, a subset of the local features can be selected, and they will affect only and exclusively the element in question. This association can be understood from two points of view. From the point of view of the model element, the association represents the functionality available in the product for that specific element. For example, if the entity `Municipality` is associated with the feature `List`, it means that this entity can be browsed in a list of the product. From the point of view of the feature, the association represents the specific configuration of the functionality. For example, if the feature `LayerManager` is

associated with an specific instance of a `Map` entity, it represents that specific map will have a layer manager. In case the element does not have a specific configuration associated with it, the default configuration will be the one specified in the global feature model. That is why in Fig. 20, within the global feature model, the features `EntityFeature`, `MapFeature` and `LayerFeature` are repeated (for space reasons their subtrees are not shown in the figure, being identical to the local feature models above them).

As this new mechanism provides more expressiveness to the SPL, it is possible to modify the initial feature model of the SPL to take advantage of the additional expressiveness. For example, the `Clustering` feature in Fig. 15 is a child of the `MapViewer` feature because it was not possible to represent that a map is composed of layers, and that some of them may be clustered if needed. With the new mechanism, we have enough expressiveness to be able to indicate that the `Clustering` feature is applied only to certain layers, and therefore the feature is a child of the feature `LayerFeature`. In addition, some features from the original feature model have been grouped depending on the elements they affect. This is why new features appear, such as `EntityFeature`, `LayerFeature` and `MapFeature` that group the functionality of entities, layers and maps respectively.

Taking this paradigm change into account, let us follow with the examples described in Section 4.1. First of all, we have two different entities, *Municipality* and *Hotel*, which require different features. Both entities have a number of properties such as *id* or *name*. Some of these properties are marked as required, so the user who creates objects of these entities must specify at least those marked as required. The decision of which properties are required and which are not is up to the domain engineer and, in this example, we have decided to mark as required only those that we consider essential when creating an object, such as its identifier and name. In addition, both entities are related to each other since a hotel is located within a municipality. Fig. 21 shows an object diagram that defines the entities of the example and the features associated to each entity from the feature model defined in Fig. 20. We can see how the *Municipality* entity is linked to a instance of the *local feature* `EntityFeature`, and some of its subfeatures are selected (`Form`, `List`, `FormAccess`, and `Filterable`), while `Creatable` and `Editable` are not selected. However, we can see that the entity `Hotel` has all the sub-features of the *local feature* `EntityFeature`.

In order to define these models, we have used a Domain Specific Language (DSL), which evolves from the one used in Alvarado et al.

D. de Castro et al.

The Journal of Systems & Software 213 (2024) 112035

(2020), that allow us to define the data model, the visualization model, and the local features of the elements that may have one. The DSL was specified with a BNF grammar and its parser was implemented with ANTLR[7]. The resulting architecture is as shown in Fig. 22. The SPL presented in this application case is based on an annotative derivation engine we developed in a previous project, and which was presented at SPLC'22 (Cortiñas et al., 2022). The derivation engine receives a JSON file with the specification of the configurations that have to be applied to the source code templates and generates the final product. This derivation engine supports the concept of multimodel, since the JSON configuration file must contain the product's feature model, but also other models such as the data model, visualization model, or navigation model (although other types of models could be used in different SPLs). This derivation engine is freely available[8]. The DSL used for the examples described is shown next.

```
1  CREATE ENTITY Municipality (
2      id Long IDENTIFIER,
3      name String DISPLAY_STRING REQUIRED,
4      geom Polygon,
5      hotels Hotel RELATIONSHIP(1..1, 0..*)
          BIDIRECTIONAL
6  ) WITH FEATURES (Form, List, FormAccess, Filterable);
7
8  CREATE ENTITY Hotel (
9      id Long IDENTIFIER,
10     name String DISPLAY_STRING REQUIRED,
11     stars Integer,
12     capacity Integer,
13     geom Point,
14     municipality Municipality RELATIONSHIP MAPPED_BY
          hotels
15 ) WITH FEATURES (Form, Creatable, Editable, List,
        FormAccess, Filterable);
```

Listing 1: Definition of entities for the WebEIEL product

Listing 1 shows the definition of the entities *Municipality* (L1) and *Hotel* (L8). For both, a series of properties are defined (such as *id, name* or *capacity*) and, at the end of the definition, the features that are associated to each entity are indicated by means of the statement WITH FEATURES.

The example of the *MunicipalityMap* and the *HotelsMap* can be seen in Fig. 23 and in Fig. 24. The former shows an object diagram that defines a simple map viewer that displays the municipalities. The map contains two layers: the municipalities and a base layer (e.g., the OpenStreetMap[9] tiles to show the map context). This map viewer requires no features, since it provides few functionality to the user (i.e., the user cannot change the layers and cannot use the geolocation feature).

```
1  CREATE GEOJSON LAYER municipalitiesLayer AS
        Municipalities FOR Municipality
2  WITH STYLES (
3      blueColor DEFAULT
4  );
5
6  CREATE MAP municipalitiesMap AS Municipalities map
        WITH LAYERS (
7      baseLayer IS_BASE_LAYER DEFAULT_BASE_LAYER,
8      municipalitiesLayer
9  ), WITH CENTER [ [40.712, -74.227], [40.774, -74.125]
        ];
```

Listing 2: Definition of the municipalities layer and map

Listing 2 shows the definition of the municipalities map in the DSL. It first defines the *municipalitiesLayer* (L1), just setting its name and a style[10]. Then, it defines (L6) the *municipalitiesMap* that shows the different municipalities of the province using the layer *municipalitiesLayer*. As we can see, there is not mention to any feature since none is required.

Fig. 24 shows an object diagram that defines a more complex map viewer, *HotelsMap*, that displays both hotels and municipalities. The map contains three layers: the hotels, the municipalities and a base layer. The map viewer provides much more functionality to the user than *MunicipalitiesMap*. In this case, the user can change the layers, and use the geolocation feature to zoom the map to his/her current position. Similarly, the hotels layer also provides more functionality (i.e., the user can change the style and the geographic objects are clustered to avoid cluttering the map at low scales).

```
1  CREATE GEOJSON LAYER hotelsLayer AS Hotels FOR Hotel
2  WITH STYLES (
3      starsStyle DEFAULT,
4      capacityStyle
5  );
6
7  CREATE MAP hotelsMap AS Hotels map WITH LAYERS (
8      baseLayer IS_BASE_LAYER DEFAULT_BASE_LAYER,
9      municipalitiesLayer,
10     hotelsLayer WITH FEATURES ( StyleSelector,
          Clustering )
11 ), WITH CENTER [ [40.712, -74.227], [40.774, -74.125]
        ]
12 WITH FEATURES ( LayerManager, UserGeolocation );
```

Listing 3: Definition of the hotel layer and map

Listing 3 shows the definition of the hotels map in the DSL. Line 1 it defines the *hotelsLayer* with multiple styles that represent the hotels depending of their number of stars or its capacity (*starsStyle* and *capacityStyle* respectively). Then, the *HotelsMap* is defined containing a base layer, the municipalities, and the hotels (L7). The *hotelsLayer* is linked to the features StyleSelector and Clustering (to be able to switch between its different styles, and to cluster the objects to avoid cluttering the map). Besides that, the *HotelsMap* is linked to the features LayerManager and UserGeolocation.

```
1  CREATE GIS WebEIEL WITH FEATURES (TopMenu,
        UserManagement);
```

Listing 4: Definition of the WebEIEL product with global features

Finally, we have to define the features that are global to the product. Listing 4 shows an example that selects the TopMenu feature and the UserManagement feature from the feature model shown in Fig. 15.

This case study shows that our proposal allows a much deeper level of customization when defining maps and layers because the features are assigned to each of them individually. It is important to remark that this level of customization is possible because some of the features of the feature model have been associated with parts of the product by means of the DSL. Furthermore, in our implementation of the SPL, a product does not include every time all the features and only activates them for the selected entities/elements. The features will only be included if there is an entity or element that has it selected. For example, if no entity of the model has selected the forms feature (i.e., Form), the source code associated with this feature will not be included in the product.

The complete object diagram of the example can be seen in Fig. A.25 (that can be found in Appendix). The object diagram represents the result of instantiating entities, maps and layers with the DSL to generate the WebEIEL product. The diagram shows how each element has a series of associated features independent of the others, as it has been explained throughout this section.

---

[7] www.antlr.org
[8] https://github.com/AlexCortinas/spl-js-engine
[9] OpenStreetMaps website: https://www.openstreetmap.org/

[10] The visualization model includes styles for the layers, which have been omitted for clarity in the excerpts and explanations.
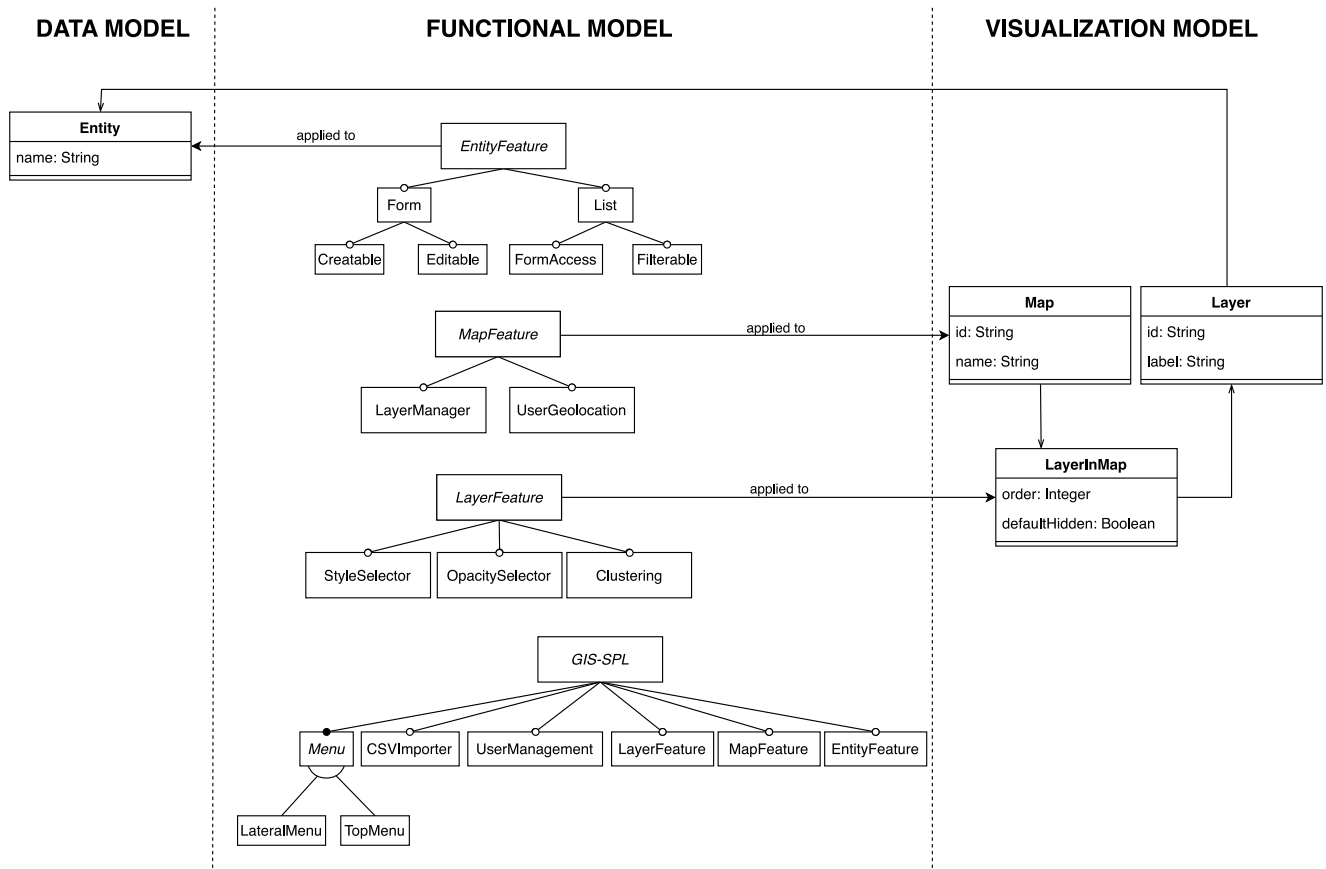
Fig. 20. Diagram representing the integration of the features with the concrete elements of the application.

4.3. Discussion

Next, we present a summary of the application of local features to entities, maps, and layers. The data model of WebEIEL consists of 107 entities:

- 17 entities are used to represent indicators computed from the data and they are just pairs of (administrative region, indicator value). Therefore, these entities do not have an associated form (i.e., these entities do not require the feature `Form`) and they are only listed applying the feature `Filterable`.
- 11 entities represent the context of the maps (e.g., provinces, municipalities, hidrography, etc.). These entities have an associated form (feature `Form`), and hence they are listed applying the local features `FormAccess` and `Filterable`. However, since their values must not be modified, visualizing them as in a non-editable form was the best decision and the local features `Creatable` and `Editable` are not applied.
- The other 79 entities have associated forms with the features `Creatable` and `Editable`, and are listed with the `FormAccess` and `Filterable` features.

WebEIEL also includes 54 maps. 44 of them do not need the features `LayerManager` nor `UserGeolocation` because they display province-wide indicators with a single layer where geolocating the user is not useful. The remaining 8 maps need the feature `LayerManager` and `UserGeolocation` because they use multiple layers and show detailed data. The product has 150 layers. In this case, the feature `Clustering` was not applied to any of those layers, the feature `OpacitySelector` was applied to all of them, and the `StyleSelector` was applied to some of the layers depending on the particular needs of each case.

These numbers show the high degree of customization that was necessary to apply to the entities, maps, and layers of the system. The case study shows how the concept of local features was applied in a real, non-trivial SPL. The description of the case study shows that the concept of local feature appears in many parts of this SPL, and the resulting solution shows the feasibility and adequacy of the concept of local features for modeling and specifying variability in systems where some features must be bound to specific elements of the system specified in other system's models. As we have explained in the study of alternatives presented in the introduction, addressing this requirement with existing variability modeling proposals would result in solutions that we consider unsatisfactory, either because they lead to artificially complex feature models or because they lead to bad design decisions.

5. Conclusions and future work

SPLs have become a relevant paradigm in the development of families of software products, with many success cases in industrial settings. SPLs reduce development costs and increase product quality by allowing us to semi-automatically develop software products from a set of core assets that are adapted and customized based on the selection of features that must be present in the product we want to generate. In this article, we presented the concept of local feature, which brings the possibility of associating features to the system's elements to which they should be applied in the application engineering phase, that is, when the product to be generated is defined and configured. This allows us to apply features to custom system elements, and not just to pre-established variation points decided at the domain engineering phase. Local features allow us to further customize the products generated with the software platform, which consequently allows us to better meet software requirements, as we have seen in many examples throughout the article.
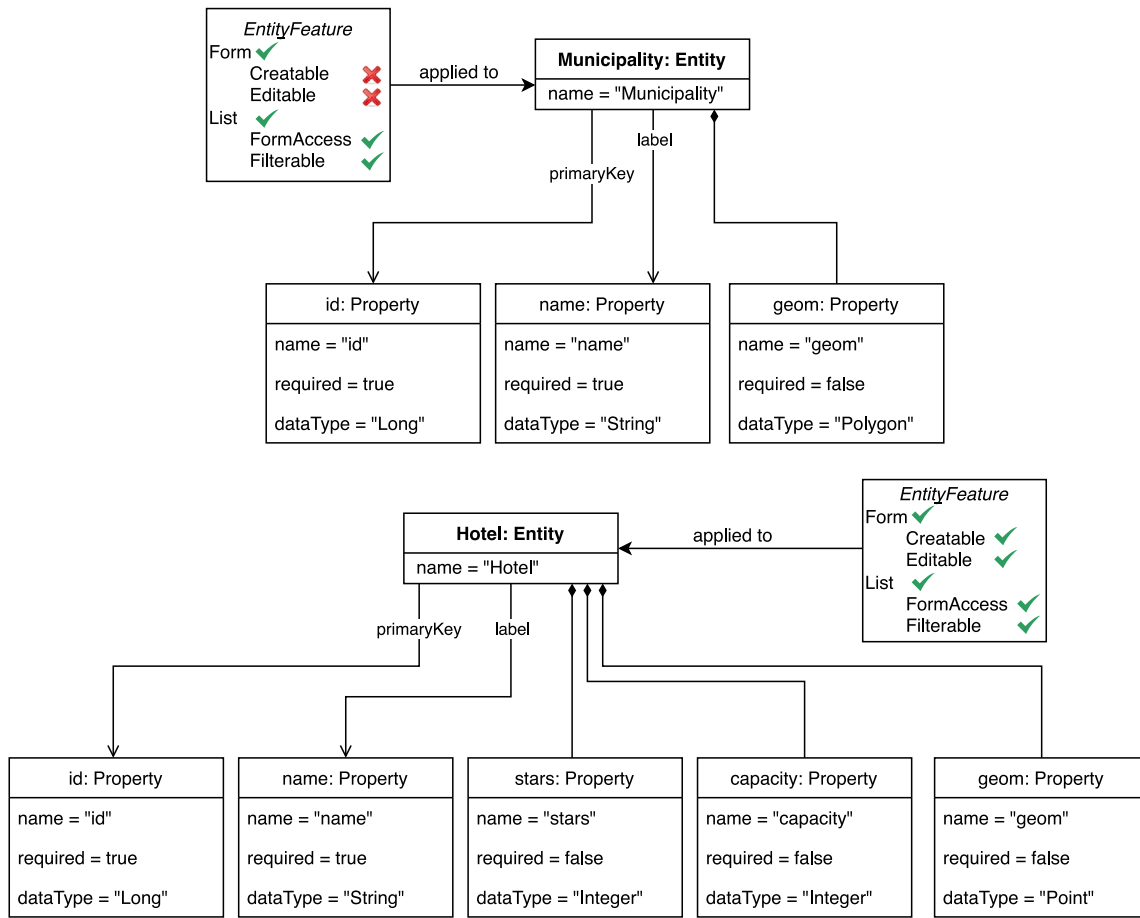
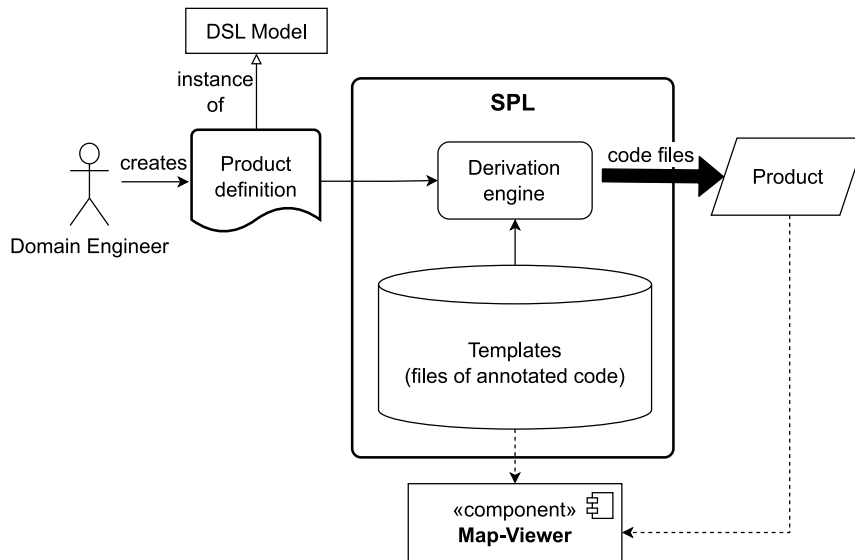**Fig. 21.** Object diagram of the entities.

Fig. 22. Architecture diagram.

The article develops the concept of local features to propose a practical implementation. In our proposal, the feature model comprises two types of features: global and local. Global features are either selected or not selected when configuring a product and, if selected, they are applied at a predefined variation point (in the domain engineering phase). In the case of local features, if they are selected for a new product, they must be associated to the system's elements to which they must be applied. Therefore, the variation points affected by a local feature are decided at the application engineering phase, that is, when we define and build a new product. Since we need to associate local features to other system's elements, we model local features using the concept of multimodel, which allows us to establish associations
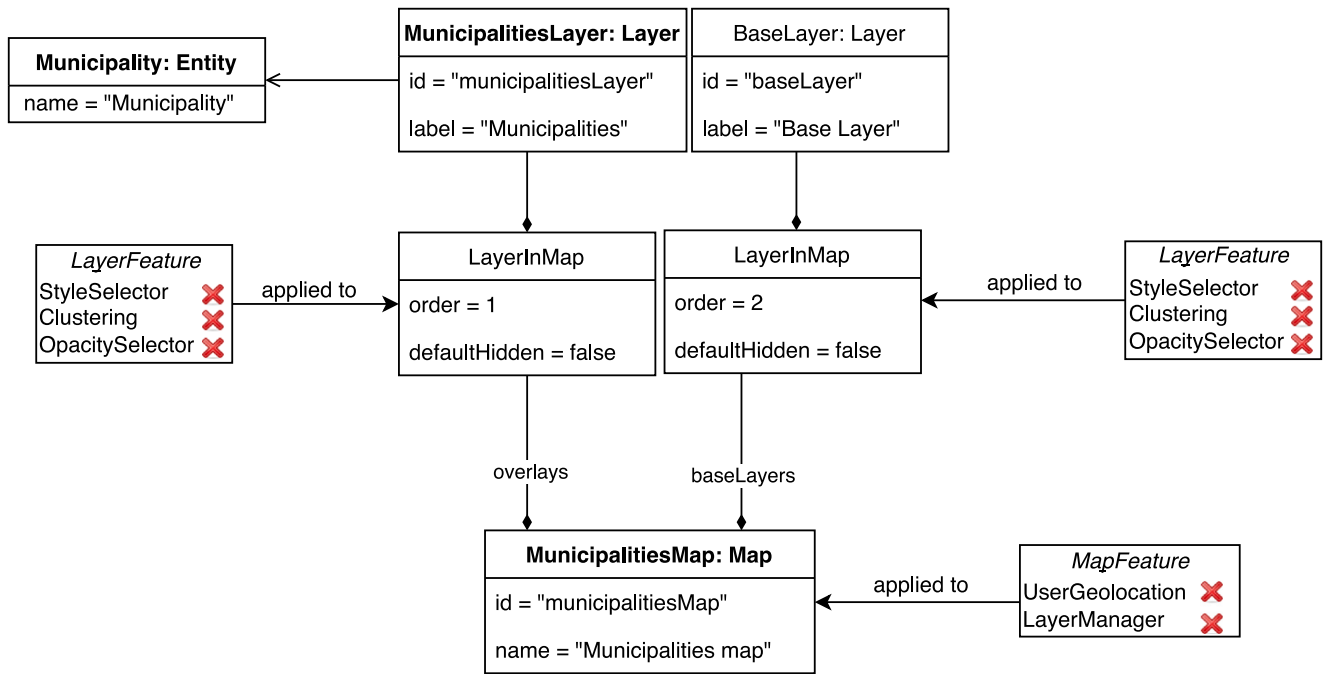
Fig. 23. Object diagram of the municipalities map.
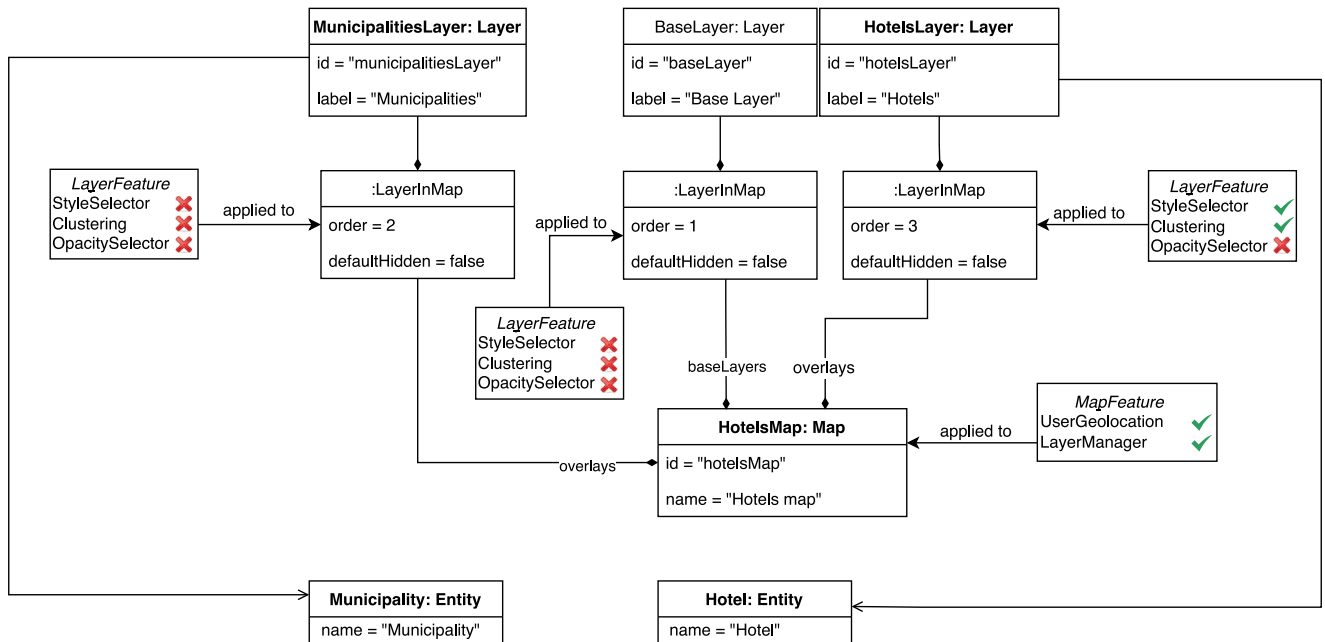

Fig. 24. Object diagram of the hotels map.

between features and elements of other system's models, such as the data model. We have provided different examples that illustrate the usefulness of the concept of local features in real scenarios.

In order to demonstrate the usefulness and benefits of this approach, we applied our proposal to an SPL for GIS applications. One of the problems to be solved regarding the implementation of local features is the mechanism to define their associations with other system's elements. We proposed a domain specific-language to define these associations. In the application case, we showed how local features could be used to customize the application of the features regarding data visualization in maps based on the application requirements. The visualization model of a GIS defines which maps the users will see and how the information

regarding entities with a geo-spatial component is organized into layers that will be part of those maps. The definition of the visualization model is not standard, that is, each GIS has its own visualization model depending on the requirements (for example, the visualization models would be different for a GIS for road management than that of a GIS for electricity supply management). This allowed us to decide in the application engineering phase which features would be applied to each elements of the visualization model. This application case allowed us to show that our proposal allows us to achieve a higher level of customization of the products generated with the SPL.

The application case we have presented in Section 4 shows how local features were implemented with a DSL. This DSL was implemented
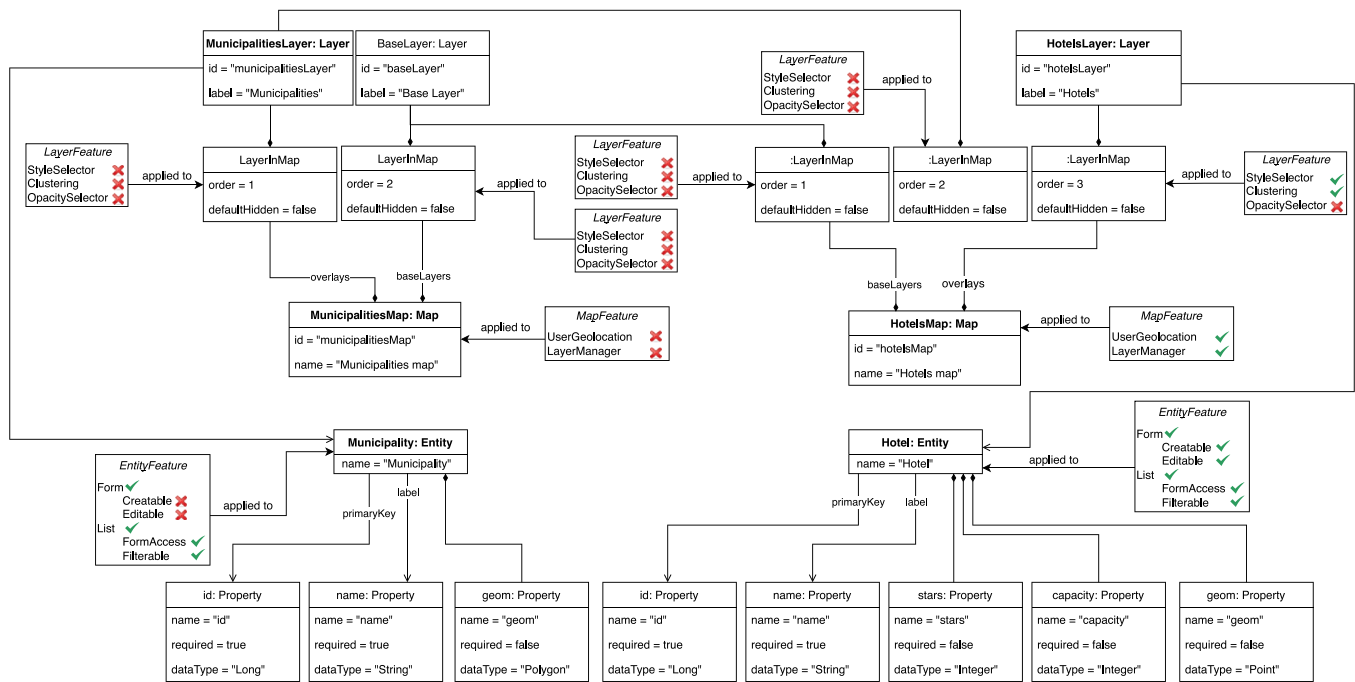
Fig. A.25. Excerpt of WebEIEL object diagram.

as part of a real project and it is, therefore, specific of the domain of GIS (since we considered that in this way it would be easier to use). However, as future work, we are considering the design of a more generic domain specific language that allows us to specify the associations between local features and elements from other system's models.

CRediT authorship contribution statement

David de Castro: Conceptualization, Methodology, Software, Validation, Writing – original draft, Writing – review & editing. **Alejandro Cortiñas:** Conceptualization, Formal analysis, Investigation, Methodology, Resources, Software, Validation, Writing – original draft, Writing – review & editing. **Miguel R. Luaces:** Conceptualization, Formal analysis, Funding acquisition, Investigation, Methodology, Resources, Software, Supervision, Validation, Writing – original draft, Writing – review & editing. **Oscar Pedreira:** Conceptualization, Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Resources, Software, Supervision, Validation, Writing – original draft, Writing – review & editing. **Ángeles Saavedra Places:** Conceptualization, Formal analysis, Funding acquisition, Investigation, Methodology, Resources, Validation, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Alejandro Cortnas Alvarez, Miguel Angel Rodriguez Luaces, Oscar Pedreira, Angeles Saavedra Places reports financial support was provided by Spain Ministry of Science and Innovation. David de Castro Celard, Alejandro Cortnas Alvarez, Miguel Angel Rodriguez uuaces, Oscar Pedreira, Angeles Saavedra Places reports financial support was provided by Xunta de Galicia.

Data availability

No data was used for the research described in the article.

Acknowledgments

Appendix. WebEIEL object diagram

See Fig. A.25.

References

Alférez, M., Acher, M., Galindo, J.A., Baudry, B., Benavides, D., 2019. Modeling variability in the video domain: language and experience report. Softw. Qual. J. 27 (1), 307–347. http://dx.doi.org/10.1007/s11219-017-9400-8.

Alvarado, S.H., Cortiñas, A., Luaces, M.R., Pedreira, O., Places, A.S., 2020. Developing web-based geographic information systems with a DSL: proposal and case study. J. Web Eng. 167–194. http://dx.doi.org/10.13052/jwe1540-9589.1923.

Apel, S., Batory, D., Kästner, C., Saake, G., 2016. Feature-Oriented Software Product Lines. Springer.

Apel, S., Kästner, C., 2009a. An overview of feature-oriented software development. J. Object Technol. 8 (5), 49–84. http://dx.doi.org/10.5381/jot.2009.8.5.c5.

Apel, S., Kästner, C., 2009b. An overview of feature-oriented software development. J. Object Technol. 8 (5), 49–84.

Barkmeyer, E., Barnard, A., Denno, P., Flater, D., Libes, D., Steves, M., Wallace, E., 2003. Concepts for Automating Systems Integration. Tech. Rep. NISTIR 6928, NIST - National Institute of Standards and Technology, http://dx.doi.org/10.6028/NIST.IR.6928.

Batory, D., Benavides, D., Ruiz-Cortes, A., 2006. Automated analysis of feature models: Challenges ahead. Commun. ACM 49 (12), 2–3. http://dx.doi.org/10.1145/1183236.1183264.

Benavides, D., Segura, S., Ruiz-Cortés, A., 2010. Automated analysis of feature models 20 years later: A literature review. Inf. Syst. 35 (6), 615–636. http://dx.doi.org/10.1016/j.is.2010.01.001.

Benavides, D., Trinidad, P., Ruiz-Cortés, A., 2005. Automated reasoning on feature models. In: Proceedings of the 17th International Conference on Advanced Information Systems Engineering. (CAiSE'05), pp. 491–503. http://dx.doi.org/10.1007/11431855_34.

Berger, T., Collet, P., 2019. Usage scenarios for a common feature modeling language. In: Proceedings of the 23rd International Systems and Software Product Line Conference-Volume B. pp. 174–181.

Cortiñas, A., Luaces, M.R., Pedreira, Ó., 2022. Spl-js-engine: A JavaScript tool to implement software product lines. In: Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume B. SPLC'22, ACM Press, pp. 66–69. http://dx.doi.org/10.1145/3503229.3547035.

Cortiñas, A., Luaces, M.R., Pedreira, O., Places, A.S., Perez, J., 2017. Web-based geographic information systems SPLE: Domain analysis and experience report. In: Proceedings of the 21st International Systems & Software Product Line Conference. SPLC'17, vol. 1, pp. 190–194. http://dx.doi.org/10.1145/3106195.3106222.

Czarnecki, K., Bednasch, T., Unger, P., Eisenecker, U., 2002. Generative programming for embedded software: An industrial experience report. In: Proceedings of the International Conference on Generative Programming and Component Engineering. GPCE'02, vol. LNCS 2487, Springer-Verlag, pp. 156–172. http://dx.doi.org/10.1007/3-540-45821-2_10.

Czarnecki, K., Helsen, S., Eisenecker, U., 2004. Staged configuration using feature models. In: Proceedings of the International Conference on Generative Programming and Component Engineering. GPCE'04, vol. LNCS 3154, Springer, pp. 266–283. http://dx.doi.org/10.1007/978-3-540-28630-1_17.

Czarnecki, K., Helsen, S., Eisenecker, U., 2005a. Formalizing cardinality-based feature models and their specialization. Softw. Process: Improv. Pract. 10 (1), 7–29. http://dx.doi.org/10.1002/spip.213.

Czarnecki, K., Helsen, S., Eisenecker, U., 2005b. Staged configuration through specialization and multilevel configuration of feature models. Softw. Process Improv. Pract. 10 (2), 143–169. http://dx.doi.org/10.1002/spip.225.

de Castro, D., Cortiñas, A., Luaces, M.R., Pedreira, Ó., Places, Á.S., 2022. Improving the customization of software product lines through the definition of local features. In: Proceedings of the 26th ACM International Systems and Software Product Line Conference (SPLC'22) - Volume a. ACM, pp. 199–209. http://dx.doi.org/10.1145/3546932.3547006.

Galindo, J.A., Benavides, D., 2020. A python framework for the automated analysis of feature models: A first step to integrate community efforts. In: Proceedings of the 24th ACM International Systems and Software Product Line Conference. SPLC'20, vol. B, pp. 52–55. http://dx.doi.org/10.1145/3382026.3425773.

González-Huerta, J., Insfran, E., Abrahão, S., 2012. A multimodel for integrating quality assessment in model-driven engineering. In: Proceedings of the 8th International Conference on the Quality of Information and Communications Technology. QUATIC'12, pp. 251–254. http://dx.doi.org/10.1109/QUATIC.2012.14.

Gonzalez-Huerta, J., Insfran, E., Abrahão, S., 2013. Defining and validating a multimodel approach for product architecture derivation and improvement. In: Proceedings of the International Conference on Model-Driven Engineering Languages and Systems. MODELS'13, vol. LNCS 8107, Springer, pp. 388–404. http://dx.doi.org/10.1007/978-3-642-41533-3.

Iglesias, A., Iglesias-Urkia, M., López-Davalillo, B., Charramendieta, S., Urbieta, A., 2019. Trilateral: Software product line based multidomain IoT artifact generation for industrial CPS. In: Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development. MODELSWARD'19, pp. 64–73. http://dx.doi.org/10.5220/0007343500640073.

Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S., 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. Rep. CMU/SEI-90-TR-021, Software Engineering Institute, pp. 1–161.

Karataş, A.S., Oğuztüzün, H., Doğru, A., 2013. From extended feature models to constraint logic programming. Sci. Comput. Program. 78 (12), 2295–2312. http://dx.doi.org/10.1016/j.scico.2012.06.004.

Metzger, A., Pohl, K., Heymans, P., Schobbens, P.-Y., Saval, G., 2007. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In: 15th IEEE International Requirements Engineering Conference. (RE 2007), IEEE, pp. 243–253.

Pohl, K., Böckle, G., Linden, F.V.D., 2005. Software Product Line Engineering: Foundations, Principles and Techniques. Springer, p. 467. http://dx.doi.org/10.1007/3-540-28901-1.

Riebisch, M., Böllert, K., Streitferdt, D., Philippow, I., 2002. Extending feature diagrams with UML multiplicities. In: Proceedings of the 6th World Conference on Integrated Design & Process Technology (IDPT2002). pp. 1–7.

Rincon, L., Rodriguez, G., Martinez, J.C., Alvarez, G.I., Pabon, M.C., 2015. Creating virtual stores using software product lines: An application case. In: Proceedings of the 10th Computing Colombian Conference. CCC'15, pp. 71–78. http://dx.doi.org/10.1109/ColumbianCC.2015.7333414.

Sharp, D., 1998. Reducing avionics software cost through component based product line development. In: Proceedings of the 17th DASC. AIAA/IEEE/SAE. Digital Avionics Systems Conference, vol. 2, pp. G32/1–G32/8. http://dx.doi.org/10.1109/DASC.1998.739846.

Siegmund, N., Ruckel, N., Siegmund, J., 2020. Dimensions of software configuration: On the configuration context in modern software development. In: Procs. of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020). ACM Press, pp. 338–349. http://dx.doi.org/10.1145/3368089.3409675.

Sousa, G., Rudametkin, W., Duchien, L., 2016. Extending feature models with relative cardinalities. In: Proceedings of the 20th International Systems and Software Product Line Conference. SPLC'16, pp. 79–88. http://dx.doi.org/10.1145/2934466.2934475.

Trujillo, S., Batory, D., Diaz, O., 2007. Feature oriented model driven development: A case study for portlets. In: Proceedings of the 29th International Conference on Software Engineering. ICSE'07, pp. 44–53. http://dx.doi.org/10.1109/ICSE.2007.36.

Voelter, M., Visser, E., 2011. Product line engineering using domain-specific languages. In: Proceedings of the 15th International Software Product Line Conference. SPLC'11, pp. 70–79. http://dx.doi.org/10.1109/SPLC.2011.25.

Weiss, D.M., Clements, P., Krueger, C.W., 2006. Software product line hall of fame. In: Proceedings of the 10th International Software Product Line Conference. SPLC'06, p. 237. http://dx.doi.org/10.1109/SPLINE.2006.1691614.

Weiss, D.M., Lai, C.T.R., 1999. Software Product-Line Engineering - A Family-Based Software Development Process. Addison-Wesley.

David de Castro is an Ph.D. in computer science student at the Database Lab of the University of A Coruña (Spain). He graduated in M.Sc. in Computer Science from the University of A Coruña in 2021 and he started his research journey during his final project, entitled "Domain Specific Language for Geographic Indicator Visualization Applications", where he entered into the world of model-driven development and Geographic Information Systems. His research topics of interest are geographic information systems, software product lines and application lifecycle management.

Alejandro Cortiñas is an Assistant Professor at the Database Lab of the Universidade da Coruña (Spain). He received a Ph.D. in Computer Science from the same university in 2017 for his thesis, entitled "Software Product Line for web-based Geographic Information Systems". His research topics of interest include software product lines, generative programming, geographic information systems, and spatial big data.

Miguel R. Luaces received his M.S. degree in Computer Science from the University of A Coruña (Spain) in 1998 and a Ph.D. in Computer Science from the same university in 2004. He undertook research in the area of spatial, temporal and Spatio-temporal databases at the FernUniversität Hagen (Germany) under the ChoroChronos project funded by the European Union. Today, he is an Associate Professor at the University of A Coruña, and he is currently the coordinator of the Databases Laboratory of the University of A Coruña. His research interests include Geographic Information Systems, Spatio-temporal Databases, Software Engineering, and Web-based Information Systems.

Oscar Pedreira has an M.Sc. and Ph.D. degree in Computer Science from the University of A Coruña, Spain. He is an Associate Professor since 2008 at the same institution. He is a researcher of the Database Laboratory. His research interests include topics in databases (algorithms for similarity search, data structures and algorithms for graph databases, geographic information systems), and in software engineering (process improvement, testing, MDE, and SPL). He has co-authored many articles published in journals and conferences relevant to the research areas mentioned. He has continuously participated in research projects and technology and knowledge transfer projects with different companies.

Angeles Saavedra-Places is currently an Associate Professor at the Computer Science Department of the University of A Coruña. She received her Ph.D. in Computer Science in 2003 from the same university. Her research interests are in the areas of Digital Humanities, Web Information Systems, Geographic Information Systems and Software Engineering. For further details about her CV see: http://lbd.udc.es/ShowResearcherInformation.do?id=5.