Improving the Customization of Software Product Lines through the Definition of Local Features

David de Castro[*]
Universidade da Coruña
Centro de Investigación CITIC
Laboratorio de Bases de Datos
A Coruña, Spain
david.decastro@udc.es

Alejandro Cortiñas[*]
Universidade da Coruña
Centro de Investigación CITIC
Laboratorio de Bases de Datos
A Coruña, Spain
alejandro.cortinas@udc.es

Miguel R. Luaces[*]
Universidade da Coruña
Centro de Investigación CITIC
Laboratorio de Bases de Datos
A Coruña, Spain
luaces@udc.es

Óscar Pedreira[*]
Universidade da Coruña
Centro de Investigación CITIC
Laboratorio de Bases de Datos
A Coruña, Spain
oscar.pedreira@udc.es

Ángeles Saavedra Places[*]
Universidade da Coruña
Centro de Investigación CITIC
Laboratorio de Bases de Datos
A Coruña, Spain
asplaces@udc.es

ABSTRACT

Variability in software product lines (SPL) is mostly described with feature models. In basic feature models, the selection of a feature for a particular product determines whether or not the feature is present in the product in a global manner. Even though there are cardinality-based feature models that allow a subset of features to be specified a number of times for each product, it is not possible to customize each instance of the feature with specific details for different elements of the product.

Some SPLs integrate model transformations and use domain specific languages to describe elements of the application that cannot be described using features (for example, the definition of the data model for a particular product). In this context, a stakeholder may require some features to be applied to some elements of the data model, but not globally (for example, not every entity in the data model may require an edition form). However, current feature models do not allow the stakeholder to specify this information.

In this paper, we propose a solution that solves this problem using domain-specific languages. In addition to defining global features for the entire application, our proposal allows the stakeholder to define local features that are specific to some elements such as parts of the application or specific entities of the data model and, using the DSL to define the product, those local features can be assigned to these elements or entities. This specification of the scope of application of features opens the door to a higher degree of customization of the generated products, thus improving their quality.

KEYWORDS

Software Product Line, variability, feature model, Domain Specific Language

[*]All authors have contributed equally and the names are listed in alphabetical order

1 INTRODUCTION

Software Product Lines (SPL) meant a great advance in the development of product families because the automatic generation of these products led to a considerable reduction of time and cost in their development. This is the reason why there are more and more SPL oriented to very different application domains [26], from the generation of software for air planes [22] to those that focus on the generation of software for IoT devices [15]. If we analyze the use of SPL in the field of web applications, these are rather scarce, although we find some examples such as the generation of web portals, virtual stores, or digital libraries [19, 21, 24]. However, with the rise of web technologies in recent years, it is expected that more and more SPLs will be oriented to this field.

The main part of any SPL is the feature model [5], a hierarchical tree that represents all the functionalities or capabilities that appear in a product family. When creating a new product within a SPL, the software engineer selects which features from the feature model will be included in it. In this way, it is possible to generate software that only contains those functionalities that are necessary. For example, all e-commerce stores are very similar to each other, but each one has different functionalities: some allow payment by credit card, others with PayPal, and others with both options.

However, there are domains in which selecting functionalities is insufficient to generate a product because the products also require the description of a data model. The product family of web-based Geographic Information System (GIS) applications is a paradigmatic example of an application domain in which it is possible to define a software product line that needs to be complemented with a data model. The standardization process carried out by the Open Geospatial Consortium (OGC) and International Organization for Standardization (ISO) has defined a set of standards for GIS that are currently followed by most software libraries. Hence, web-based GIS applications are very similar to each other and they share functional features and most of the technologies. The main difference between two web-based GIS applications is the data model that defines the set of entities that may be displayed in the map viewer, and the visualization model (i.e., layers, styles and maps) that defines the way in which the entities are displayed. For example, a GIS product for a parcel company requires a map viewer showing the drivers, warehouses and roads, while a GIS product oriented to promote the tourism in a region requires a map viewer showing hotels, attractions or nature landscapes. In the case of this product family, it is necessary that the user can define the data model that will be generated in the product.

For some years now, the Database Laboratory has been working on a software product line that generates web-based GIS applications; this 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 [7]. The functionalities are modelled 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. The software product line has been used industrially for the definition of several products (e.g., heritage management, facility management, reduced-mobility accessibility, or public transport management). We have been able to observe that, despite considerably reducing the time-to-market, the products generated have always presented a similar problem. The functionality defined by selecting features in the feature model is always applied in a general way and it is no possible to customize it in a particular way for different cases. For example, the functionality available in the map viewer of the application is selected once and it is the same for all the maps of the generated product, when the most usual thing is that the product requires maps of different types with different functionalities and that show different data. This causes the need for an adaptation effort of each product that must be carried out by software developers and that separates the product from the family of products.

In this paper, we present the mechanism that we have defined in our software product line to deal with this problem. The mechanism consists of defining features at two levels: global, which apply to the entire product; and local, which apply only to certain elements of the data model. We achieve this by creating dependencies between data model elements and feature model features. To achieve this, we follow an approach already seen when creating software product lines, the use of Domain Specific Languages (DSL), but doing it in such a way that allows the user to establish these relationships that have been previously discussed. The rest of the paper is organised as follows. Section 2 provides an overview of our previous work, which explains the context and the problem we found, the starting point for our proposal. Section 3 explains our proposal, conceptually, with an brief example, and showing the DSL grammar. Section 4 presents a case study to illustrate our proposal with an existing GIS web application, showing an example step by step. Section 5 briefly introduces the state of the art regarding feature modelling, and contrast it with our specific context. Finally, Section 6 concludes the paper and comments some ideas for future work.

2 DEVELOPING GEOGRAPHIC INFORMATION SYSTEMS WITH A DOMAIN SPECIFIC LANGUAGE

A simplified excerpt of the feature model for our web-based GIS SPL [7] is shown in Figure 1, containing a set of features that will be used throughout the article to discuss the problem and present the running example, while a small data model for the example is shown in Figure 2 with two entities, *Municipality* and *Hotel*.

The left-most branch defines features related to the entities defined in the data model (`EntityFeature`). `Form` generates detailed views of the entities. Depending on the sub-features `Creatable` and `Editable`, these detailed views provide the possibility to create or edit new elements of the entities. This is, selecting all the sub-features from `Form`, the generated application includes a component to view the details of both the *municipalities* and the *hotels*, allowing the user to create new *municipalities* and the *hotels*, and to modify the details of the existing ones. The feature `List` generates listings for each entity, in which all of its elements are displayed in a paginated table. This 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`) that activate functionalities such as geolocating the user (`UserGeolocation`), managing the map layers (`LayerManager`), the options to switch the style and the opacity of a layer (`StyleSelector` and `OpacitySelector` respectively) and clustering the elements of the layer.

The last branches define features to decide whether 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`).

In order to be able to select the features and define the data model to generate GIS, a web application was implemented through which these products could be defined and generated. For the selection of features, the user was provided with a feature tree through which the user could select those that would be present in the generated product and, for the definition of the domain, the user was provided with a UML class diagram editor, since this is easy to understand and widely adopted for data modelling.

Once this first version of the SPL started to be used with real products, some problems started to appear that had not been contemplated so far. In the first version of the SPL only one map was generated for the application, and in this map all the geographic data of all the entities were inserted. Hence, when there were many data from different entities, they overlapped one on top of the other and the interaction with them became very complicated, besides the fact that having just one map with all the geographic data is not the desired functionality for most GIS applications. Furthermore, GIS applications require the definition of a visualization map (i.e., layers, styles and maps) that are related to the entities of the data model. Although we contemplated the possibility of extending the UML class diagram editor to allow the definition of the visualization model together with the data model, we concluded that it would made the diagram difficult to understand and maintain.

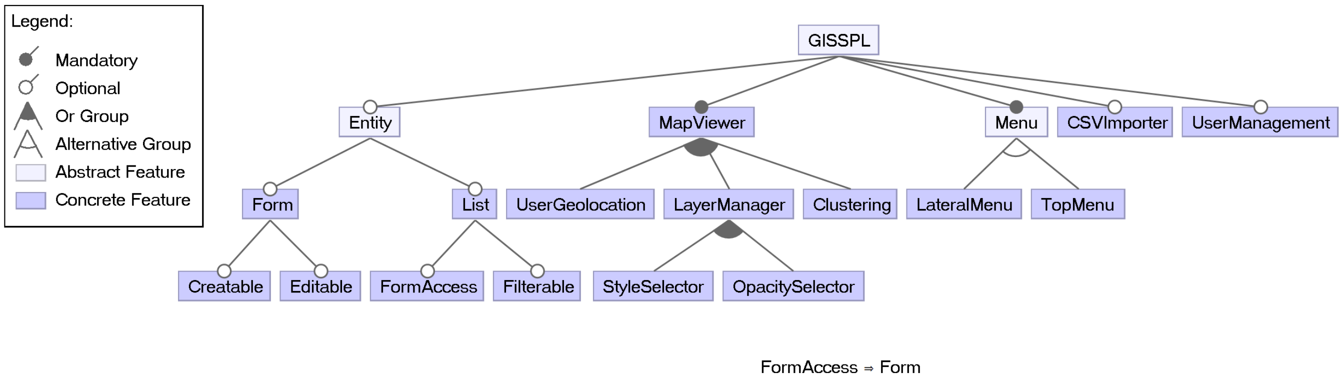

Figure 1: Simplified excerpt of the SPL feature model [7]

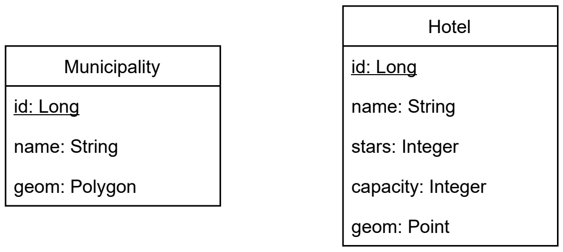

Figure 2: Example of a simple data model for a web-based GIS

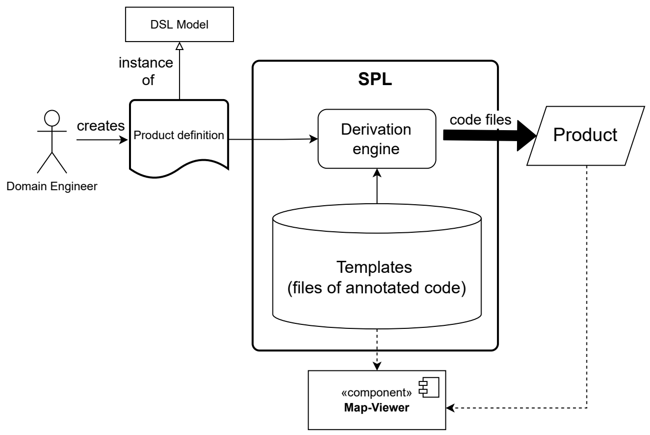

Figure 3: Architecture diagram

Therefore, we decided to create a Domain Specific Language (DSL) that allows the user to define a product through it and that, besides being able to select the features and define its data model, would also allow defining a visualization model, i.e. maps, layers and styles. In order to solve the problem of having a single map, the creation of the DSL brought with it the possibility of being able to define elements beyond the entities, so the definition of maps, layers and styles was implemented. Maps are nothing more than a grouping of layers, which has one or more base layers (that serve as background) and a series of overlays that represent geographic data of the entities. The layers are those that represent the geographic data of an entity and can be of different types: GeoJSON, in which the data is recovered by a standard REST service in textual format (GeoJSON format[1]), and represented directly on the map; WMS, in which the data comes from a map server such as GeoServer[2], and published in the map viewer as images through a Web Map Service[3] data are represented by a geographic data server; and TileLayers, similar to the WMS layers but provided by external map servers.

All these changes were addressed in a previous work [2]. A DSL was designed and created through which the products are defined. Figure 3 shows the resulting architecture diagram after the creation of the DSL. In the left side of the figure we can see the DSL, which was implemented using Antlr4[4] within a NodeJS[5] application. In the center of the figure, there is the SPL, implemented with our own engine, spl-js-engine[6], also within a NodeJS application. The products generated are web applications that have both a server, developed in Java, and a web client, developed with VueJS[7] and JavaScript.

Figure 4 shows a simplified excerpt of the meta model of the DSL[8]. A set of relevant statements of the DSL, which are used in the examples of this paper, are listed and briefly explained below:

- `CREATE GIS`: identifies a new product. The *name* of the product shall be indicated.
- `CREATE ENTITY`: defines a new entity for the product. Each entity requires a *name* and a series of attributes. Each attribute itself has its name, a data type, information of whether the attribute is required or not, etc. The relationships with other entities are also indicated.

[1]GeoJSON website: https://geojson.org
[2]GeoServer website: https://geoserver.org
[3]Web Map Service Standard - Open Geospatial Consortium webiste: https://www.ogc.org/standards/wms

[4]Antlr4 website: https://www.antlr.org/
[5]NodeJS website: https://nodejs.org/en/
[6]spl-js-engine Github repository: https://github.com/AlexCortinas/spl-js-engine
[7]VueJS website: https://vuejs.org
[8]Several elements have been omitted to simplify the examples (e.g., the model used to express relationships between entities). Full details of the DSL can be found in [2].

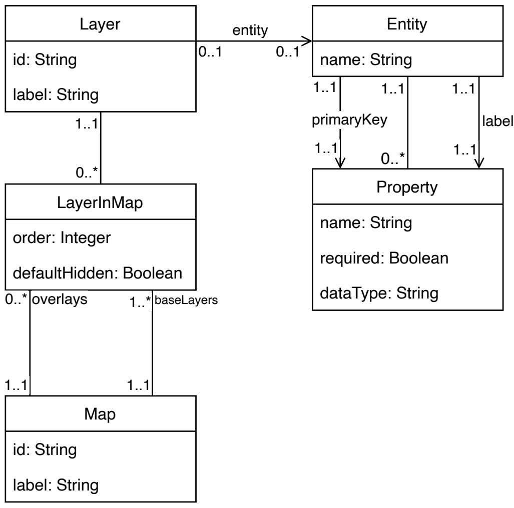

Figure 4: Simplified excerpt of the DSL meta model

- `CREATE GEOJSON STYLE`: defines a new style for a GeoJSON layer, indicating the fill and border colours as well as its opacity.
- `CREATE GEOJSON LAYER`: defines a GeoJSON layer that can be included in any of the defined maps. A GeoJSON layer is the representation of the geographic information of a particular entity of the data model; therefore, the entity shall be indicated when creating the layer. The style is also set, applying one of the previously defined (using `CREATE GEOJSON STYLE`).
- `CREATE MAP`: defines a new interactive map viewer, choosing which of the created layers are displayed in it. For each one of them it is possible to indicate a default style. Some layers are used as base layers of the map viewer (this is, they serve as background of the map), whereas others are overlays.

Thanks to these modifications, the degree of customization of the products has increased, being able to define several maps for every product, each one with its own set of layers and styles. It is important to note that this DSL has progressively changed over time and, as it has been used in projects, modifications have been made to improve it.

However, it is important to note that the use of DSLs in Software Product Lines for product configuration is not a new idea. It has been previously used in SPLs as a means to define features with cardinality and attributes. This solution, as explained in [25], is currently being used by real products (e.g., to define a user interface consisting of a menu and the elements it contains).

In these products, the use of features with cardinality and attributes through a DSL allows the stakeholder to achieve greater customization of the generated products, something that would not be possible using only a feature model. This problem is very similar to the one in our case, we need a tool that allows us to achieve greater customization when defining the products. The creation of the DSL solved a large part of the problem by allowing us to define concrete elements of the application, such as maps and layers, but it was still not possible to associate features with these types of elements, so that when a feature was selected from the model, it affected the entire application. For example: if the user selected the feature `Form`, all the entities of the model would be affected and would have a detail form. Even more, in our example in Figure 2, it is quite likely that we want the users of our application to be able to update the details of a hotel, or to create new hotels; however, allowing the users to create or modify municipalities is very undesirable, since municipalities are not changing over time, and allowing them to be edited can only increase the risk of accidentally modifying your data. The same occurs with specific elements of the application, such as maps and layers: if the feature `LayerManager` is selected, all maps would have a layer manager even though these do not require it.

Following the same example as before, let us imagine that in the application we have two maps: on the one hand we have a map that represents the different municipalities of the province and, as they are always represented in the same color, it is not necessary to select the feature `StyleSelector` for the layer and, therefore, `LayerManager` is not needed neither. On the other hand, we have a map for the representation of the different hotels in the province, which can be represented with different colors depending on their number of stars or their capacity. That is why for the layer that represents the hotels it is mandatory to select the feature `StyleSelector` and, for the map, the feature `LayerManager` since it would be possible to change the style of the hotels layer to represent them depending on their stars or capacity.

It was previously mentioned that there are solutions in the industry that make use of cardinality-based feature models with attributes. However, for this case it would not be a viable alternative. It may be possible to use cardinality and attributes to define the classes and attributes of an UML class diagram in the feature model. However, the result would be difficult to understand because feature models describe functionality and not data models. Furthermore, there is no tool that supports cardinality-based feature model, and we would have to implement it from scratch instead of reusing existing functionality.

Therefore, the objective of this publication is to allow, not only global features that affect the whole application, but also to allow the user to select features that locally affect specific elements or entities of the application, increasing significantly the customization of the product generated by the SPL.

3 INTEGRATING FEATURE MODELS INTO A DSL

Considering that a product in our system is defined by a feature selection in a feature model (to select the functionality of the product) and an instance of a DSL (to define the data model and the visualization model), we must define a mechanism to integrate both artifacts (i.e., the feature selection and the DSL instance) in such a way that some features may be selected multiple times and each selection supports complex customization. As an example, a GIS application may have three different map viewers (i.e., `MapViewer` must be selected three times), each map viewer may show a different set of layers and styles (i.e., a different visualization model must be defined for each feature), and each map viewer may have different functionality (i.e., each feature requires a different selection of child features).

Our proposal consists of defining two types of features: *global features* and *local features*. Global features are those that are selected only once to define the global functionality of the product. Local features are those that can be defined multiple times, can be configured individually, and are associated with specific elements of the DSL. Figure 5 describes our proposal in more detail. In the center of the figure, there is a subset of some elements of DSL (i.e., `Property`, `Entity`, `Layer`, `LayerInMap`, and `Map`). In our proposal, we allow DSL elements to be associated with fragments of the feature model. This association can be understood from two points of view. From the point of view of the DSL 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 `MapViewer` is associated with an specific instance of a `Map` entity, it represents the specific map that will be displayed in the map viewer.

Figure 5 shows a model of all the local features that we have currently defined in our SPL. A DSL *Entity* is related with an `EntityFeature` feature that describes the functionality that is available in the product for the data model *Entity* (i.e., whether there is form or a list for the *Entity*, and which actions are available in each case). Each map viewer available in the product is described by a DSL *Map* element that is associated to a `MapFeature` feature that describes the functionality available in the map viewer (i.e., whether the user can manage the layers in the map or they are fixed, or whether the map offers geolocation of users). Finally, each occurrence of a given layer in a map is represented by a *LayerInMap* element associated to a `LayerFeature` feature that describes the functionality of the layer in the map (i.e., whether the user can modify the style or the opacity of the layer, or whether the geographic objects are clustered to avoid cluttering the map).

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 Figure 1 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 Figure 5.

```
CREATE ENTITY <name> (
    <propertyName> <dataType> [ IDENTIFIER | REQUIRED |
      DISPLAY_STRING | UNIQUE ]
    <referenceName> <entity> RELATIONSHIP (<c1>..<c2>, <c3>..<c4>)
      [ BIDIRECTIONAL | MAPPED BY ] <referenceNameInTheOtherEntity>
) WITH FEATURES ( [ Form | Creatable | Editable | List | FormAccess
      | Filterable | ... ] );
```

Listing 1: Definition of an entity

Listing 1 shows a fragment of the DSL grammar to define entities of the data model (i.e., the element *Entity* in Figure 5). For each entity, the user defines its properties as well as the references to other entities (which will be omitted throughout the paper to simplify the problem, since they do not contribute anything to the practical case). In addition, with the `WITH FEATURES` statement, the user defines a list of features affecting only the entity in question, thus managing to deglobalize the product.

```
CREATE GEOJSON LAYER <name> AS <displayName> FOR <entityName>
      [EDITABLE]
WITH STYLES (
    <styleName1> [DEFAULT],
    <styleName2> [DEFAULT],
    ...
);
```

Listing 2: Definition of a GeoJSON layer

Listing 2 shows a fragment of the DSL grammar to define maps in the product (i.e., the element *Map* in Figure 5). For each layer, the user defines the layer's data source and the available styles.

```
CREATE [SORTABLE] MAP <name> AS <displayName> WITH LAYERS (
    <layerName1> [ IS_BASE_LAYER | DEFAULT_BASE_LAYER | HIDDEN ]
      [style] WITH FEATURES ([ StyleSelector | OpacitySelector |
      Clustering | ... ]),
    <layerName2> [ IS_BASE_LAYER | DEFAULT_BASE_LAYER | HIDDEN ]
      [style] WITH FEATURES ([ StyleSelector | OpacitySelector |
      Clustering | ... ]),
    ...
)
WITH CENTER [LatLngBounds | {"lat": Double, "len": Double, "zoom":
      Integer} | Integer]
WITH FEATURES ([ UserGeolocation | LayerManager | ...]);
```

Listing 3: Definition of a map

Listing 3 shows a fragment of the DSL grammar to define maps in the product (i.e., the element *Map* in Figure 5). For each map, the user defines the list of layers that are available in the map, as well as the characteristics of each of them, which includes the features associated with each one of them. Last, the user can define the specific functionalities of that map. The reason why the features are assigned to the layers when inserting them into a map is because in this way it is possible to assign, to the same layer, different features depending on the needs of the map in which it is going to be inserted. If they were related to the layer, the customization capacity would be reduced.

```
CREATE GIS <product-name> WITH FEATURES ([ TopMenu | LateralMenu |
      CSVImporter | UserManagement | ... ]);
```

Listing 4: Definition of the product with global features

Finally, although the selection of features at the local level is important to solve the problems discussed above, the DSL must still be able to allow the selection of features that affect the entire product. As an example of these global features are those related to the menu (`LateralMenu` and `TopMenu`), data import from CSVs (`CSVImporter`), or user management (`UserManagement`). These features are not related to any specific element, so they are selected when the product is defined with the statement shown in Listing 4.

It is important to remark that, in addition to defining the DSL, we also had to modify the annotated code of the SPL. Previously, it

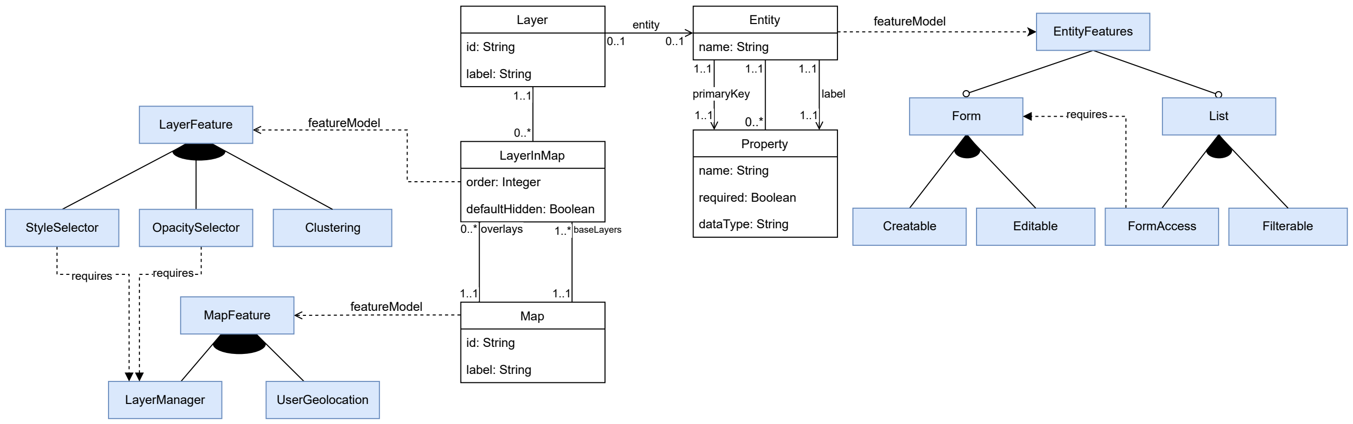

Figure 5: Diagram representing the integration of the features with the concrete elements of the application

was only necessary to to take into account whether a feature from the feature mode was selected or not (i.e., the feature where global). Now, a feature may be selected by specific elements of the data model (i.e., the features are local). Thus, it was necessary to modify the control mechanisms of the SPL derivation engine to support this new functionality.

Thanks to this new approach, it is possible to achieve a higher degree of customization in complex products that require a higher level of detail. This new approach opens the possibility of specifying features to specific entities, maps and layers.

4 CASE STUDY

We will illustrate how our proposal works with a case study. WebEIEL[9] is a web application dedicated to publish geographic information collected by the provincial council of A Coruña (Spain). The web application which is currently available was developed in 2008-2010 and it uses outdated technology. The last year, we started a project to update WebEIEL to use current technology and we decided to use our software product line. The WebEIEL data model is simple because it 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. At the same time, it is extensive (because it includes around 100 hundred entities). Furthermore, there are two different types of entities: those that can be modified (e.g., a water treatment plant), and those that are static and do not change (e.g., a municipality). Finally, there are two types of maps: full-fledged map viewers that display a large collections of layers and provide complex functionality to the user, and simple map viewers that display few layers (sometimes only one) and provide almost no functionality to the user. An example of the first type of map would be a *water supply map* that includes as layers the *water sources*, the *water distribution system*, and the *water treatment plants*, and that provides the user functionality to change the order and the style of the layers, zoom the map in and out, clustering the geographic objects to avoid cluttering the map, etc. An example of the second type of map would be an indicator map that shows the municipalities of the province coloured according to their population density. This map would include a single layer (municipalities) with a single style and with few functionality (for example, zooming and panning is not necessary).

[9]https://webeiel.dacoruna.gal

The product generated with our SPL with the mechanism that we propose in this paper had much less functionality than the original WebEIEL application. Given that the SPL did not allow to customize the map viewer, if a layer manager was needed for a map that contained several layers (by selecting the *LayerManager* feature), it was also applied to others that only had one layer and did not need it. Thus, a developer had to modify the product to create different types of map viewers. Furthermore, the SPL did not allow to define a product with several map viewers. Hence, a product with a single map viewer containing all the layers had to be defined, and a developer had to modify the product to create the different map viewers.

A similar problem occurred in relation to the functionality available for the entities of the data model. An entity such as *park* requires a form because it has a large collection of properties describing each park (i.e., the feature `Form`) and it may be edited by a user because the properties change often (i.e., the feature `Editable`). However, the entity *municipality* does not require a form because the only property is the name, and it is never edited by users because the municipalities rarely change, and when they do the modifications are so complex that they cannot be performed in the web application. Just like the previous case, given that the SPL did not allow to select different features for each entity, the product had to be generated with the `Form` and the `Editable` features and a developer had to modify the product to avoid the municipalities to be editable.

In these examples, we can clearly see the problem of assigning features globally to the entire product. In the examples that follow, we will describe how these problems can be solved with our proposal. Instead of showing the DSL instance for the complete WebEIEL application, we will show a small example with the two entities shown in Figure 2. As explained above, the entity *Municipality* cannot be created nor edited, whereas the entity *Hotel* can be edited and created. Figure 6 shows an object diagram that defines the entities shown in Figure 2 and the features associated to each entity from the feature model defined in Figure 5 (using green

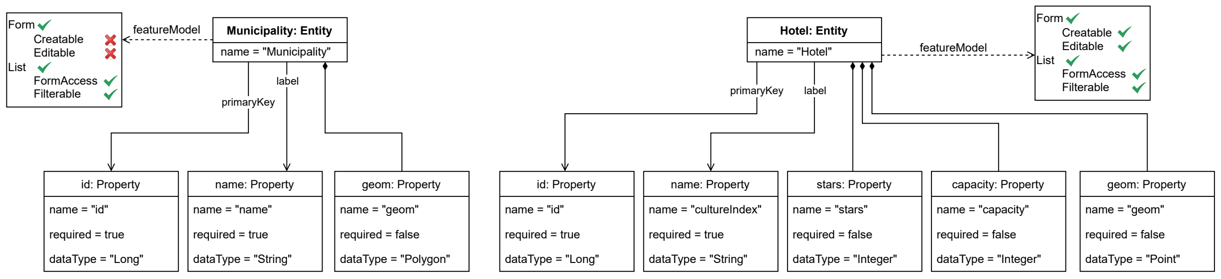

Figure 6: Object diagram of the entities

checks for the selected features and red crosses for the unselected ones).

```
CREATE ENTITY Municipality (
    id Long IDENTIFIER,
    name String DISPLAY_STRING REQUIRED
) WITH FEATURES (Form, List, FormAccess, Filterable);

CREATE ENTITY Hotel (
    id Long IDENTIFIER,
    name String DISPLAY_STRING REQUIRED,
    stars Integer,
    capacity Integer
) WITH FEATURES (Form, Creatable, Editable, List, FormAccess,
      Filterable);
```

Listing 5: Definition of entities for the WebEIEL product

Listing 5 shows the definition of the entities *Hotel* and *Municipality*. 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`. While for hotels almost all the features are selected (such as `Form`, `List` or `Creatable`), for the case of municipalities only some of them are selected and others like `Editable` and `Creatable` are omitted since, unlike hotels that are susceptible to be modified some of their properties (such as the number of stars or their capacity), municipalities rarely change, so it is not necessary to have forms for their creation and modification.

Figure 7 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 tiles to show the map context). The map viewer provides few functionality to the user (i.e., the user cannot change the layers and cannot use the geolocation feature). Similarly, the layers also provide few functionality (i.e., the user cannot change the style or the opacity, and the geographic objects are not clustered).

```
CREATE GEOJSON LAYER municipalitiesLayer AS Municipalities FOR
      Municipality
WITH STYLES (
    blueColor DEFAULT
)
WITH FEATURES ();

CREATE MAP municipalitiesMap AS Municipalities map WITH LAYERS (
    baseLayer IS_BASE_LAYER DEFAULT_BASE_LAYER,
    municipalitiesLayer
), WITH CENTER [ [40.712, -74.227], [40.774, -74.125] ]
WITH FEATURES ();
```

Listing 6: Definition of the municipalities layer and map

Listing 6 shows the definition of the municipalities map in the DSL. It first defines the *municipalitiesLayer* that has no features selected because the style and opacity cannot be changed, and no clustering is required. Then, it defines the *municipalitiesMap* that shows the different municipalities of the province using the layer *municipalitiesLayer*. Given that the map does not allow the user to change the style, it does not have features.

Figure 8 shows an object diagram that defines a complex map viewer that displays the hotels and the 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 the map defined in Figure 7 because the user can change the layers and can 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).

```
CREATE GEOJSON LAYER hotelsLayer AS Hotels FOR Hotel
WITH STYLES (
    starsStyle DEFAULT,
    capacityStyle
)
WITH FEATURES (StyleSelector, Clustering);

CREATE MAP hotelsMap AS Hotels map WITH LAYERS (
    baseLayer IS_BASE_LAYER DEFAULT_BASE_LAYER,
    hotelsLayer
), WITH CENTER [ [40.712, -74.227], [40.774, -74.125] ]
WITH FEATURES ( LayerManager, UserGeolocation );
```

Listing 7: Definition of the hotel layer and map

Listing 7 shows the definition of the hotels map in the DSL. It defines the *hotelsLayer* with multiple styles that represent the hotels depending of their number of stars or its capacity (*starsStyle* and *capacityStyle* respectively). Finally, the definition selects the features `StyleSelector` and `Clustering` (to be able to switch between its different styles, and to cluster the objects to avoid cluttering the map). Then, the *hotelsMap* is defined containing the hotels as well as the municipalities. As the layer *hotelsLayer* has a style selector, it is mandatory to select the `LayerManager` feature in order to access the corresponding selector (as expressed in the restriction defined in the feature model shown in Figure 5).

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

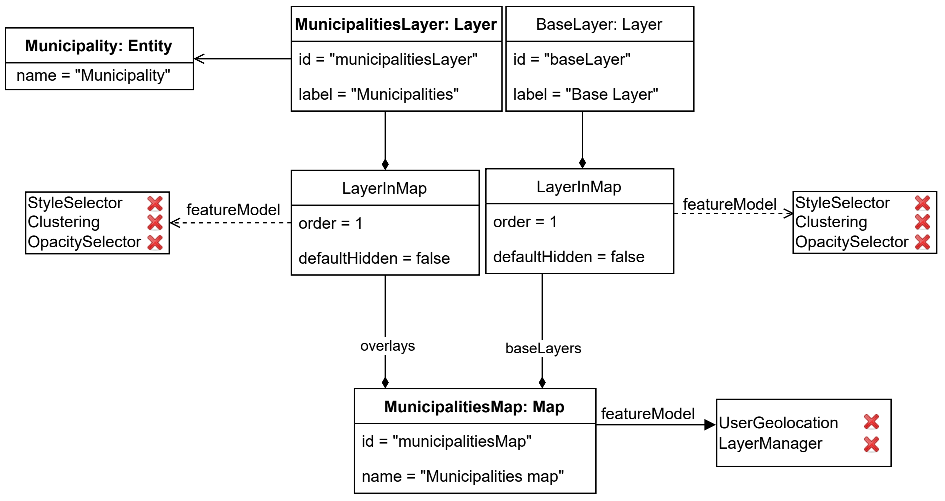

Figure 7: Object diagram of the municipalities map

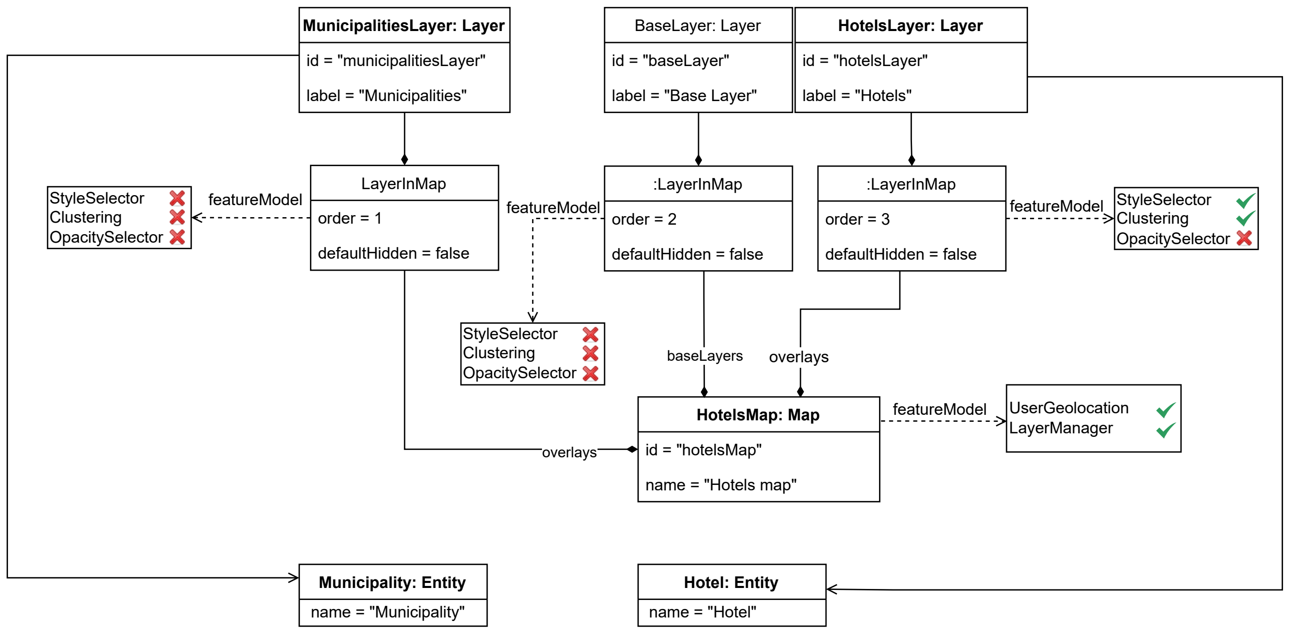

Figure 8: Object diagram of the hotels map

Listing 8: Definition of the WebEIEL product with global features

Finally, we have to define the features that are global to the product. Listing 8 shows an example that selects the `TopMenu` feture and the `UserManagement` feature from the feature model shown in Figure 1.

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. For example, while the hotels layer has a style selector, the municipalities layer does not, as it is unnecessary since it has only one style.

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 Figure 9 (that can be found in Section A). The object diagram represents the result of instantiating entities, maps and layers with the DSL to generate the WebEIEL product. In it, it can be seen how each element has a series of associated features independent of the others, as it has been explained throughout this section.

5 RELATED WORK

Feature modelling [16] is the "de facto" variability representation for software product lines [3, 5, 12, 23]. A feature model is a tree where the features (end user-visible characteristics of a software system [16]) 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 [18]. 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, within a feature model a set of cross-tree constraints can be defined, for example, `including feature A implies that feature B is also included`. This description corresponds to what is called a basic feature model, but there are several extensions mentioned in the literature [1, 5].

Cardinalilty-based feature models al. [8–10, 20] allow defining a 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 [4, 6, 8, 11, 25] 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 [8, 11, 17]. This could allow for enough flexibility as to model the different elements and variability existing in our SPL for web-based GIS. However, there are several downsides: 1) A relational data model is like a graph. A hierarchical tree is not the best way to represent a graph. 2) There are not tools supporting SPL that generate web applications, that require fine grained variability for multiple different languages [13], nor tools supporting feature modelling with both cardinalities and feature attributes which also serve to implement a product line and generate products[10] [14]. 3) As explained in Section 2, the need to define local features appears when we already have a SPL that uses both a feature model and a DSL to specify products. It made much more sense to keep using our DSL, since it was already implemented and working, and not starting over.

6 CONCLUSIONS AND FUTURE WORK

Software Product Lines are becoming more and more relevant in the world of software development. This has led to the appearance of increasingly sophisticated SPLs with a higher degree of customization: they no longer only have a feature model, but also support the definition of data models and specific elements of the generated applications. However, these elements were developed separately, so there was no way to assign features to specific elements of the product.

This article has proposed a solution that allows SPLs that have defined entities or elements of the application to assign features to each of these elements individually, so that not all the entities or elements of the application have the same functionalities, thereby significantly increasing the degree of customization of the SPL.

In order to demonstrate its usefulness and the benefits of this new approach, modifications were made to an SPL oriented to the generation of GIS applications, modifying a Domain Specific Language that allowed the definition of products, so that, with this new approach, it is possible to assign features to the entities defined in the data model as well as to the elements of the visualization model, such as maps and layers. To test it, a real product called WebEIEL was generated with which it was demonstrated that this new approach allows to achieve a greater customization of the generated product being able to apply features only to those parts of the application that really need them, so that the quality of the generated product increases considerably.

As a future work we will update the existing web application (see Section 2) to support the DSL. This is, besides allowing the user to select the product features and define its data model through a UML diagram, also allows the user to define it through a DSL editor. Both the DSL, the features and the UML diagram would be synchronised with each others, so the changes made in the DSL would be reflected in the UML diagram and vice versa, and the selection of features would be reflected in the DSL model and vice versa. This would offer the user two ways to create the product model according to his needs and preferences.

Additionally, another important improvement to be made is the implementation of constraint propagation in the DSL between feature models related to different elements. For example, when one of the layers of a map has the feature `StyleSelector` or `Opacity-Selector` selected, the map must have the feature `LayerManager` assigned, since this feature activates the layer manager and is the one that allows access to the style and opacity selector. It would be very useful to define a mechanism to propagate the selection of features through the constraints so that, when a feature is selected, those on which it depends are also selected automatically.

Another update we are working on is the possibility to have a default selection of features for different elements that belong to a hierarchy structure. For example, we could determine that the most likely scenario for the product line is that the feature `Clustering` would be selected for most layers. Them, we can define that the feature `Clustering` is selected by default for the *GISSPL*. But then, for a specific *Map* of a specific product, maybe the features is not desired, so the software engineer in charge of configuring the product needs to "unselect" the feature `Clustering` for this *Map*. The same situation happens with the element *Layer* and *LayerInMap*. This is, we have a hierarchy structure (*GISSPL*, *Map*, *Layer*, *LayerInMap*) in which the features may be defaulted selected or "unselected". This is not so simple to implement taking into account the possible constraints, for example.

[10]According to [14], *pure::variants*[11] supports feature attributes and partially supports cardinalities in features, and also can be used to implement a SPL and generate products, but it is a commercial tool.
[11]*pure::variants* website: https://www.pure-systems.com.

ACKNOWLEDGMENTS

Partly funded by: MCIN/AEI/10.13039/501100011033, NextGenerationEU/PRTR, FLATCITY-POC: PDC2021-121239-C31; MCIN/AEI/10.13039/501100011033 EXTRACompact: PID2020-114635RB-I00; GAIN/Xunta de Galicia/ERDF CEDCOVID: COV20/00604; Xunta de Galicia/FEDER-UE GRC: ED431C 2021/53; MICIU/FEDER-UE BIZDEVOPSGLOBAL: RTI-2018-098309-B-C32; MCIN/AEI/10.13039/501100011033 MAGIST: PID2019-105221RB-C41.

REFERENCES

[1] Mauricio Alférez, Mathieu Acher, José A. Galindo, Benoit Baudry, and David Benavides. 2019. *Modeling variability in the video domain: language and experience report.* Vol. 27. 307–347 pages. https://doi.org/10.1007/s11219-017-9400-8

[2] Suilen Alvarado, Alejandro Cortiñas, Miguel Luaces, Oscar Pedreira, and Ángeles Saavedra Places. 2020. Developing Web-based Geographic Information Systems with a DSL: Proposal and Case Study. *Journal of Web Engineering* (06 2020). https://doi.org/10.13052/jwe1540-9589.1923

[3] Sven Apel and Christian Kästner. 2009. An overview of feature-oriented software development. *Journal of Object Technology* 8, 5 (2009), 49–84. https://doi.org/10.5381/jot.2009.8.5.c5

[4] Don Batory, David Benavides, and Antonio Ruiz-Cortes. 2006. Automated analysis of feature models: Challenges ahead. *Commun. ACM* 49, 12 (2006), 2–3. https://doi.org/10.1145/1183236.1183264

[5] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Information Systems* 35, 6 (sep 2010), 615–636. https://doi.org/10.1016/j.is.2010.01.001

[6] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Automated Reasoning on Feature Models. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAiSE 2005).* 491–503. https://doi.org/10.1007/11431855_34

[7] A. Cortiñas, M. R. Luaces, O. Pedreira, A. S. Places, and J. Perez. 2017. Web-based Geographic Information Systems SPLE: Domain Analysis and Experience Report. In *Proc. 21st International Systems & Software Product Line Conference (SPLC 2017) Vol.1.* Sevilla, 190–194.

[8] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich Eisenecker. 2002. Generative programming for embedded software: An industrial experience report. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 2487 (oct 2002), 156–172. https://doi.org/10.1007/3-540-45821-2_10

[9] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2004. Staged configuration using feature models. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 3154 (2004), 266–283. https://doi.org/10.1007/978-3-540-28630-1_17

[10] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2005. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice* 10, 1 (jan 2005), 7–29. https://doi.org/10.1002/spip.213

[11] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2005. Staged configuration through specialization and multilevel configuration of feature models. *Software Process Improvement and Practice* 10, 2 (2005), 143–169. https://doi.org/10.1002/spip.225

[12] José A. Galindo and David Benavides. 2020. A Python framework for the automated analysis of feature models: A first step to integrate community efforts. *ACM International Conference Proceeding Series* Part F1644 (2020), 52–55. https://doi.org/10.1145/3382026.3425773

[13] Jose-Miguel Horcas, Alejandro Cortiñas, Lidia Fuentes, and Miguel R Luaces. 2022. Combining multiple granularity variability in a software product line approach for web engineering. *Information and Software Technology* 148 (2022), 106910.

[14] José Miguel Horcas, Mónica Pinto, and Lidia Fuentes. 2022. Empirical analysis of the tool support for software product lines. *Software and Systems Modeling* (2022). https://doi.org/10.1007/s10270-022-01011-2

[15] Aitziber Iglesias, Markel Iglesias-Urkia, Beatriz López-Davalillo, Santiago Charramendieta, and Aitor Urbieta. 2019. Trilateral: Software product line based multidomain IoT artifact generation for industrial CPS. *MODELSWARD 2019 - Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development* (2019), 64 – 73. https://doi.org/10.5220/0007343500640073 Cited by: 11; All Open Access, Hybrid Gold Open Access.

[16] Kyo C Kang, Sholom G Cohen, James a Hess, William E Novak, and a Spencer Peterson. 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. *Software Engineering Institue* 17, November (1990), 161. https://doi.org/10.1080/10629360701306050

[17] Ahmet Serkan Karataş, Halit Oğuztüzün, and Ali Doğru. 2013. From extended feature models to constraint logic programming. *Science of Computer Programming* 78, 12 (2013), 2295–2312. https://doi.org/10.1016/j.scico.2012.06.004

[18] Klaus Pohl, Günter Böckle, and Frank Van Der Linden. 2005. *Software Product Line Engineering: foundations, principles and techniques.* Vol. 49. Springer Science & Business Media. 467 pages. https://doi.org/10.1007/3-540-28901-1

[19] D. Ramos Vidal, A. Cortiñas, M. R. Luaces, O. Pedreira, and A. S. Places. 2020. A Software Product Line for Digital Libraries. In *Proc of the 16th International Conference on Web Information Systems and Technologies (WEBIST 2020).* Online, 321–334.

[20] M Riebisch, K Böllert, D Streitferdt, and I Philippow. 2002. Extending Feature Diagrams with UML Multiplicities. *6th World Conference on Integrated Design & Process Technology* (2002), 1–7. http://www.citeulike.org/group/858/article/505058

[21] Luisa Rincon, Gabriel Rodriguez, Juan C. Martinez, Gloria Ines Alvarez, and Maria Constanza Pabon. 2015. Creating virtual stores using software product lines: An application case; [Caso de Aplicación para Crear Tiendas Virtuales Usando Lineas de Productos de Software]. *2015 10th Colombian Computing Conference, 10CCC 2015* (2015), 71 – 78. https://doi.org/10.1109/ColumbianCC.2015.7333414 Cited by: 1.

[22] D.C. Sharp. 1998. Reducing avionics software cost through component based product line development. In *17th DASC. AIAA/IEEE/SAE. Digital Avionics Systems Conference. Proceedings (Cat. No.98CH36267)*, Vol. 2. G32/1–G32/8 vol.2. https://doi.org/10.1109/DASC.1998.739846

[23] Gustavo Sousa, Walter Rudametkin, and Laurence Duchien. 2016. Extending feature models with relative cardinalities. *ACM International Conference Proceeding Series* 16-23-September-2016 (2016), 79–88. https://doi.org/10.1145/2934466.2934475

[24] Salvador Trujillo, Don Batory, and Oscar Diaz. 2007. Feature Oriented Model Driven Development: A Case Study for Portlets. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on.* IEEE, 44–53. https://doi.org/10.1109/ICSE.2007.36

[25] Markus Voelter and Eelco Visser. 2011. Product Line Engineering Using Domain-Specific Languages. In *2011 15th International Software Product Line Conference.* 70–79. https://doi.org/10.1109/SPLC.2011.25

[26] David M Weiss, Paul Clements, and Charles W Krueger. 2006. Software Product Line Hall of Fame. *SPLC 2006: Proceedings of the 10th International Software Product Line Conference* (2006), 237. https://doi.org/10.1109/SPLINE.2006.1691614

A WEBEIEL OBJECT DIAGRAM

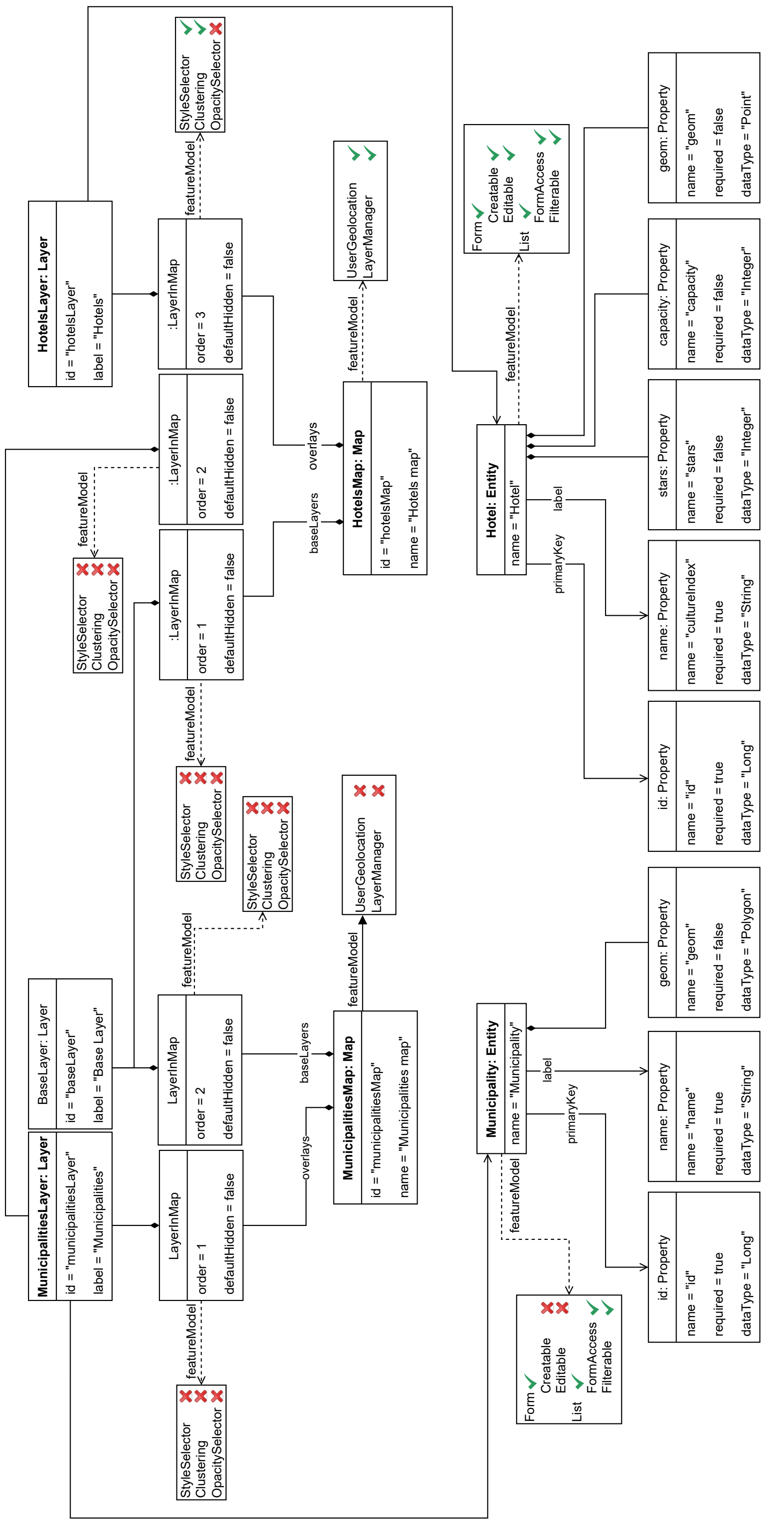

Figure 9: Excerpt of WebEIEL object diagram