An Ontology-Based Product Architecture Derivation Approach

Hector A. Duran-Limon, *Member, IEEE*, Carlos A. Garcia-Rios,
Francisco E. Castillo-Barrera, *Member, IEEE*, and Rafael Capilla, *Senior Member, IEEE*

Abstract—Software product line (SPL) engineering has proven to improve software quality and shorten development cycles, cost and time. In product line engineering, product derivation is concerned with the realization of the variability at the implementation level. However, the majority of research works focuses on instantiating the variants selected in the final product, while the derivation at the architecture level has been poorly explored. As product line engineers often customize the product architecture by hand during the application engineering phase, the derivation and customization processes of the product line architecture (PLA) might be in some cases error-prone. Consequently, in this research we present an Ontology-based product Architecture Derivation (OntoAD) framework which automates the derivation of product-specific architectures from an SPL architecture. Our solution uses a language-independent model to specify the product line architecture and a model-driven engineering approach for architecture derivation activities. We use an ontology formalism to reason about the automatic generation of model-to-model transformation rules based on the selection of features and we illustrate our approach using a voice over IP motivating example. Finally, we report results about scalability and performance regarding the size of the variability model.

Index Terms—Software product lines, feature models, software architecture, product derivation, architecture derivation, ontologies, model-driven engineering, scalability

1 Introduction

Software Product Lines (SPLs) are a successful technique being used in industry over the past 20 years to improve software quality and shorten development cycles and costs [1], [2]. An SPL focuses on systematic, planned, and strategic reuse of core assets to produce a number of products that satisfy a particular market segment [3]. Family members are often described in terms of features, where each feature represents an increment in the functionality of the product, while *Feature modeling* [4] is a technique used to specify the commonalities and variabilities of a product family. Feature models are organized in feature trees where parent and child features are described hierarchically. There are different types of relationships between a parent feature and a child feature (or subfeature) [5]. A *Mandatory* relationship states that the child is required. An *Optional* relationship means that the child may or may not be selected. An *Alternative* relationship indicates that one of the child features must be selected whereas an *Or* relationship states that one or more of the sub-features must be included. Features are logically organized around variation points, while feature constraints represent the relationships between features which are used to delimit the scope of the product portfolio.

In addition, the realization of the variability in reusable assets and products is achieved in software product lines through a derivation process. While *product architecture derivation* (also called *architecture customization process*) aims at producing a product-specific architecture from a generic product line architecture (PLA), *product derivation* is concerned with the realization of the variability of product family members at the implementation level. *Architectural variability* in an SPL architecture involves both *architectural variation points* and *architectural variants*. The variability in the feature model is often expressed in terms of variation points (i.e., an area affected by variability) and systems options known as variants. This variability can be also described in the software architecture and represented through different mechanisms.

As the derivation process is still challenging, several authors recognize the importance for automating the analysis of feature models [6] as well as the product architecture derivation process [7], [8], [9], [10], [11], [12]. In this light, we can highlight two types of derivation approaches: *copy-based* and *transformation-based*. In the former, an architecture variant is copied as defined in the SPL architecture (e.g., a UML class defined in the SPL architecture representing a variant is copied as a UML class in the product architecture) [13]. The latter involves transforming an architectural element to a different architectural element. An example of this is the transformation of a variant (i.e., an optional component) into a common architectural element (e.g. a component). This method is commonly supported by approaches using an architecture description language (ADL) as modeling language [8]. Other proposals supporting product

- H.A. Duran-Limon and C.A. Garcia-Rios are with the Department of Information Systems, University of Guadalajara, CUCEA, Mexico 45100. E-mail: {hduran, charlie}@cucea.udg.mx.
- F.E. Castillo-Barrera is with the School of Engineering, Autonomous University of San Luis Potosi, Mexico 78290. E-mail: ecastillo@uaslp.mx.
- R. Capilla is with the Department of Computer Science, Rey Juan Carlos University of Madrid, Spain. E-mail: rafael.capilla@urjc.es.

Manuscript received 6 Aug. 2014; revised 5 June 2015; accepted 16 June 2015. Date of publication 24 June 2015; date of current version 11 Dec. 2015.
Recommended for acceptance by S. Malek.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2015.2449854

0098-5589 © 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

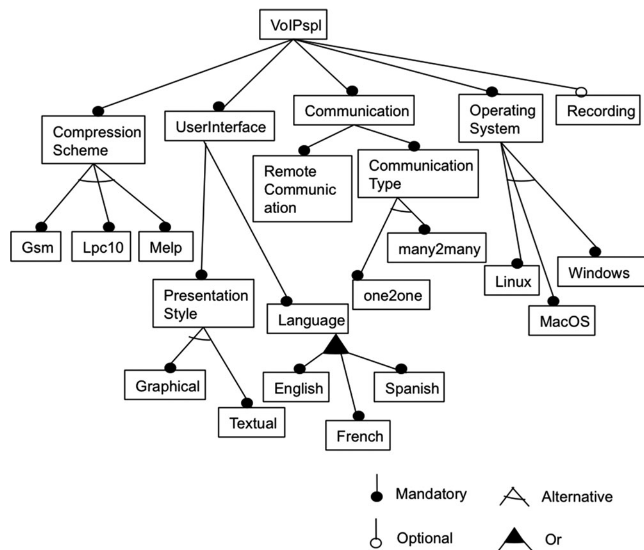

Fig. 1. Feature model tree of the VoIP system.

derivation rely on different techniques, ranging from Boolean expressions [7], [8], [9] to Model-Driven Development like [10], [14] and aspect-oriented programming (AOP) [12]. Because there are still challenges not properly solved in SPL derivation, we describe in the next section the problem statement and goals of our work.

1.1 Problem Statement

Today, the majority of the aforementioned approaches used in SPL derivation have focused on mapping feature models to model templates in order to realize the variability at the implementation level. Hence, system variants are selected and configured in the final product and described in instantiated model templates (e.g., a UML model). The majority of these approaches use propositional logic to represent features and feature constraints. Propositional formulas connect variants using logical connectors and define logic predicates that limit the values of the Boolean variables. However, the derivation at the architecture level, as a previous step before products are derived, has been poorly explored. Therefore, product line engineers often do manually the selection of the product line architecture elements and variants that will be part of a concrete product architecture. Because product line engineers may introduce errors in the mapping between features and architectural elements, we address in this work the following research challenges:

1) *Challenge 1.* Provide automatic transformation at the architecture level from a product line architecture to a product-specific architecture.
2) *Challenge 2.* Evaluate automatically the correctness of the specifications of the mappings between features and architectural elements made by the product line engineer and before the product derivation process happens.

The aforementioned challenges attempt to reduce the burden of some manual activities used in the derivation process and automate the proposed activities at a higher abstraction level than previous approaches. However, we do not address the impact of quality attributes in the architecture that may lead to structural changes during the derivation process.

The remainder of the paper is structured as follows. Section 2 presents a motivating example to exemplify the derivation process of our solution. In Section 3 we discuss the related work and their limitations, while Section 4 introduces the terminology and concepts used in this work. Section 5 outlines the OntoAD framework. In Section 6 we present the language integration phase of the framework whereas the details of the product derivation phase are described in Sections 7 to 9. In Section 10 we outline the outcome of our evaluation and we report experimental results. Finally, in Section 11 we draw the conclusions and future work.

2 Motivating Example

We use a running example to illustrate the proposed OntoAD framework throughout the paper. Our running example consists of an SPL architecture for voice over IP (VoIP) systems. The variability of the VoIP product family is shown in the feature tree of Fig. 1. For communicating voice efficiently we use different compression algorithms for data streams (e.g., GSM, LPC10, and MELP) [15], that we model as alternative features. The presentation of the user interface can be either text-based or graphical and the system supports one or several languages alternatively. Other variants in the feature model are concerned with the type of communication of VoIP systems, ranging from one-to-one or many-to-many communication types. Optional features like a conversation recording facility are also possible. The number of different possible feature configurations is 504, which is the amount of architecture products that can be derived.

The SPL architecture of the VoIP system is represented in PL-Xelha[1] as depicted in Fig. 2. As the architecture of the sender and the receiver are symmetrical, for the sake of simplicity, we only show the sender part and part of the variants and variation points. Solid lines denote the mandatory elements in the architecture while dotted lines represent the variable parts. In addition, the codec component is in charge of compressing the audio to different formats (i.e., Melp, Gsm, and LPC10), and the `c_sink_stub` component assembles the audio packets before they are sent to the network. The `c_sink_stub` performs the opposite process, i.e., unmarshalling packets received from the network.

A typical derivation process will map features to architectural elements. In our example, the feature `Recording` is mapped to the variant `c_recording`, as we can see in Fig. 2. As some other architectural elements have feature dependencies, the variant `c_melp_linux`, mapped to the feature `Melp` requires the selection of the feature `Linux`.

During the SPL architecture specification, the product line engineer may commit errors in defining the mappings between features and architectural elements. In the example of Fig. 2, if we select the features `Melp, Linux, Graphical, English, Spanish, One2one, and Recording`, and a wrong mapping is specified by the architect (e.g., `c_melp_linux requires Melp and Windows`), then the component `c_melp_linux` will not be selected during the

1. PL-Xleha is an ADL that prescribes the commonalities and variabilities of a product line and is an extension of the ADL Xelha [16].

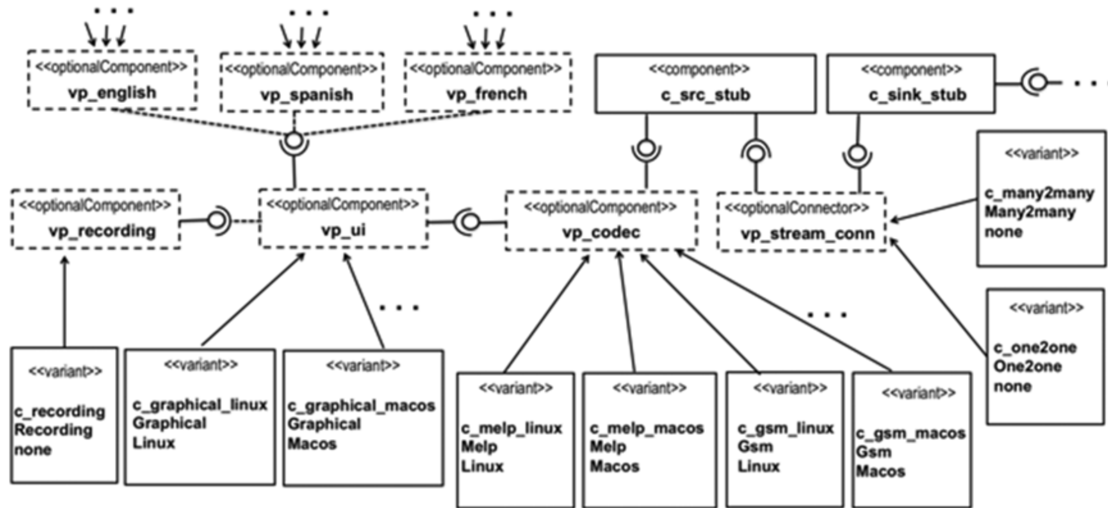

Fig. 2. SPL architecture of the VoIP System.

derivation process. Consequently, we illustrate in the following sections via our motivating example how we can ensure the correctness of the specification of the mappings before the derivation process happen.

3 Related Work

A significant number of research works have been done to realize the variability of an SPL at the implementation level. Some of the efforts that realize the variability during product derivation are: FeatureHouse [17], AHEAD [18], CaesarJ [19], Classbox/J [20], FeatureC++ [21], and Fuji [22]. Part of these approaches enable the composition of software artifacts. For instance, FeatureHouse is language-independent to enable the composition of artifacts using Java, C#, C, and XMI/UML. However, one drawback is that this approach assumes that multiple designs are predefined. In terms of its ability to incorporate different languages, FeatureHouse requires writing code to define either composition or modification rules.

Some other works have addressed the issue of deriving product architectures. Some of these approaches allow the automatic generation of component model configurations [10], [11], [12] and using ADL specifications [8], [9]. Other approaches focus on deriving UML activity and class models [7], [13], [23], [24] using propositional logic, which is recognized as a valid technique to support product derivation. Propositional logic consists in a set of sentences (i.e., a formula) where propositional variables are connected using logical connectors (i.e., AND, OR, XOR). A propositional formula encompasses a set of Boolean variables and a propositional logic predicate that constraints the values of the Boolean variables. Propositional formulas are often used to describe feature models and their associated constraints.

Czarnecki and Antkiewicz [7] suggest the so-called "fmp2rms" plug-in that uses Boolean formulas to implement the mapping between feature models and UML models (any model using MOF). The elements of the model are annotated with present conditions (PCs) and meta-expressions (MEs) which are evaluated with respect to a particular feature configuration indicating if an element should be present or removed. An extension of the plug-in has been also proposed in [25] which extends the "fmp2rms" approach to automatically pre-configure feature models based on the goals of product stakeholders and map softgoals to feature models. Similarly, the FeatureMapper approach [13] consists of a tool that defines mappings between features and model elements (e.g, software artifacts) in the problem space in order to derive the realization of the features in the solution space. In addition, the Clafer meta-modeling language [23] provides a concise notation to map feature configurations to component configurations or model templates. Clafer offers a notation to mix feature models and meta-models in the solution space during product configuration, and it provides a direct correspondence between Clafer and UML models. Finally, the Common Variability Language (CVL) [26] defines an abstraction layer and a realization layer, where fragments as MOF model instances are substituted in the base model to resolve the product model in order to compose configurable units.

In addition to the reasoning approaches based on propositional logic, other works combine ontologies with feature modeling, such as for instance Czarnecki et al. [27], where the authors cover the notational and semantic gap between both representation techniques. Peng et al. [28] use the OWL ontology to classify features into different categories and describe the dependencies and constraints between them. The authors present a reasoning-based validation mechanism for the constraints defined. Such reasoning capabilities are also mentioned in [29] as they use OWL to reason and detect conflicting knowledge in the requires and excludes constraints of feature models. These synergies have also been explored by [30], which states the difficulty of feature models to be analyzed by a reasoning tool consistently with domain knowledge. Recent works [31], [32] state also the role of ontology modeling to extend current representation capabilities of feature models by adding domain constraints to express domain concepts linked to features and zero-to-many relationships.

Finally, model-driven techniques are also used to automate part of the derivation process of product architectures, such as: [10], [11], which employ such techniques

to transform a feature model into product architectures. However, as the domain design is specified in terms of ATL transformation rules [33], the transformation processes cannot be fully automated. Other similar efforts that rely on aspect-oriented techniques [12], [24] to derive product architectures from feature selections fall out of the scope of this work.

All the aforementioned approaches constitute good attempts to provide a smooth transition from feature models to product architectures where the variability is expressed and realized. However, the derivation at higher levels of abstraction, that is from generic to concrete product line architectures, is poorly addressed. In addition, the models discussed above do not provide ways to check the correctness of the mappings between features and architecture models, as this may lead to errors during product derivation.

4 Terminology and Basic Concepts

In this Section we describe the main terminology and basic concepts of the different areas involved in our approach.

4.1 Software Architecture

There has been increasing interest in modeling variability at the software architecture level [8], [9], [34], [35], [36]. Software architecture [3] is a discipline that is emerging as a software design approach able to reach higher abstraction levels, hence, alleviating the software development complexity and making this development process less error-prone. *Architecture description languages (ADLs)* [37] represent formal notations for describing software architectures in terms of coarse-grained components and connectors. Consequently, the designer concentrates on higher-level architectural concerns whereby low-level implementation details are masked out. Commonly in ADLs the software systems are defined as an assembly of components and connectors.

4.2 Ontologies

As said earlier, we use ontologies, represented in OWL [38], to model both features and architectural semantic relationships. OWL is based on *description logic (DL)* [39], which is a family of logic-based knowledge representation formalisms. We use the Manchester OWL Syntax [38] to illustrate the OWL descriptions that are part of the running example. This syntax is mainly employed by ontology editing tools such as Protégé [38]. We use this syntax because it is easy to write and read. In addition, we use the Turtle syntax [40] to represent ontologies. The Turtle syntax is supported by most ontology-based reasoning engines. We use Turtle because its syntax is less verbose and more user-friendly than other formats. In an ontology, the domain is described in terms of *individuals*, *classes*, and *object properties*. Individuals are instances of classes and represent the basic units in the domain. Classes are sets of individuals with similar characteristics. Object properties represent binary relationships between individuals.

4.3 Metamodeling

Model-driven techniques commonly rely on the use of metamodeling [41] as a means to automate model-to-text and model-to-model transformations. In this work, we use Acceleo [42] as a model-to-text transformation language and ATL [33] as a model-to-model transformation language. Both languages are rule-based, which simplifies both the task of extracting information from a file representing a model and the task of transforming this information. A metamodel M_0 of a language *L0* describes the elements that such a language is able to use and how these elements can be used. A model defined in *L0* is made of instances of elements of the metamodel M_0. Similarly, the meta-metamodel *MM0* of a language *L0* defines the elements that the metamodel M_0 is able to use. The elements of the metamodel M_0 are instances of the elements of *MM0*. An example of metamodeling is the approach taken by the OMG to define its model-driven architecture (MDA) [43] in which the Meta Object Facility (MOF)[2] is a language used to write metamodels. For instance, UML[3] and SysML[4] are defined by MOF metamodels.

5 The OntoAD Framework

In this section we explain our approach consisting of a framework aimed to support the derivation process from a product line architecture to a customized product-specific architecture, such as Fig. 3a describes. In our framework, both the feature model and the SPL architecture model are transformed into an ontology O_{SPL} by means of the so-called *Ontology Generation Engine*. It should be noted that the module representing the Ontology Generation Engine is denoted with a dotted line. This is because this module can have different implementations, each one associated with a different SPL language. Each implementation of this module is automatically generated by the *Language Integrator Engine*. This Engine is in charge of achieving independence of the language used to define the SPL architecture.

This ontology stores information about the SPL architecture and the feature model, and it allows us to reason about the relationships between architectural elements and features. The *Ontology-based Reasoning Engine* infers the architectural elements that realize the selection of the features in terms of pair of values. To achieve this, the *Query Interface* module, using the selected features, produces a set of ontology queries that retrieves the architectural elements belonging to a particular product. Importantly, the *Query Interface* is able to detect inconsistencies in the mappings between features and architectural elements in the SPL architecture.

The *Rule Generator Engine* produces the transformation rules in charge of mapping the variability to architecture components. To achieve language independence (whereby different ADLs can be used), a *language mapping ontology*, so-called $O_{lang\ mapping}$, is used as input by the rule generator engine.

This ontology is generated automatically and contains the knowledge for mapping the meta-model M_0 of a particular SPL language and the meta-metamodel *MM0* that all

2. http://www.omg.org/mof/
3. http://www.uml.org/
4. http://www.omgsysml.org//

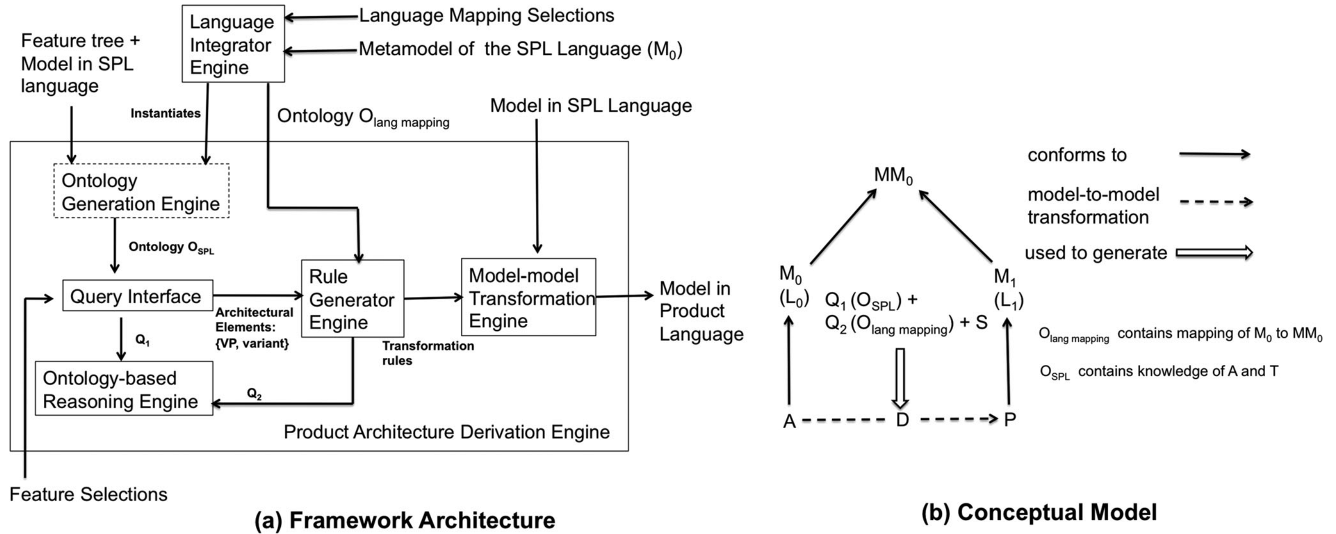

Fig. 3. The OntoAD framework activities and its conceptual model.

languages must conform. The *Model-to-model Transformation Engine*, shown in Fig. 3a, transforms the SPL architecture into a product-specific architecture.

5.1 Derivation Process

Our proposed derivation process encompasses the following three stages:

1) *Language integration*. The first stage is required to achieve language independence and is carried only one time unless more than one SPL language is used. At this stage, the meta-model M_0 of the SPL language is specified using the Ecore language [44].
2) *Domain modeling*. The second stage encompasses the domain modeling activity and involves the following sub-tasks: (i) definition of the feature model, and (ii) definition of the SPL architecture containing the mappings between features and architecture elements. For these two sub-tasks we used the visual feature tree editor suggested in [45] and a complementary tool for editing the SPL architecture.
3) *Product architecture derivation*. The third stage concerns the derivation process and consists of: (i) the selection of the variants in the feature model, (ii) the generation of model-to-model transformation rules, and (iii) the transformation of the product-specific architecture.

As we automate the generation of transformation rules, the product architecture derivation stage is simple but yet powerful. We used Pellet[5] as the Ontology-based Reasoning Engine and an ATL transformation engine [33] as the Model-to-model Transformation Engine.

5.2 The OntoAD Conceptual Model

The OntoAD conceptual model relies on the ability of automating the generation of a set of rules D able to transform an SPL architecture A into a product architecture P, as shown in Fig. 3b. The description of the OntoAD model is as follows. D is the set $\{C_R, T_R\}$, in which C_R is a set of model-to-model copy rules and T_R is a set of model-to-model transformation rules. Both P and A can be defined in different languages, given that a) L_0 is a language for defining A and L_1 is a language for defining P, b) L_1 is a subset of L_0, and c) both the metamodel M_0 of L_0 and the metamodel M_1 of L_1 conform to the meta-metamodel MM_0 (see Section 4.3 for basic concepts regarding metamodeling).

The generation of D rules is carried out based on the information provided by both the ontology $O_{lang\ mapping}$ and the ontology O_{SPL}, together with the feature selections S of a feature tree T. The ontology $O_{lang\ mapping}$ represents the mapping of the elements of a metamodel M_0 to the elements of a meta-metamodel MM_0. The ontology O_{SPL} contains information about an architecture A and a feature tree T, given that the elements in A are architectural instantiations of features in T. A set of queries Q_1 (including information about the feature selections S) are performed over the ontology O_{SPL} to produce a set of tuples t of the form *{VP, Variant}*. Tuples represent architectural variation points and their selected architectural variants, respectively. Given that a selected feature f is an instantiation in A, there is a many-to-one relationship between f and VP.

The information obtained by a set of the queries $Q2$ using the ontology $O_{lang\ mapping}$ together with tuples t are in charge of generating both a set of model-to-model copying rules C_R and a set of model-to-model transformation rules T_R. The former are used to copy the mandatory elements while the latter realize the variability into common architectural elements.

Finally, we used one-to-one transformations to simplify the model, as it involves transforming or copying a single architecture element defined in A to a single architectural element defined in P. In the following section we present details of the language integration phase.

6 Integrating an ADL to OntoAD

In order to achieve language-independence, the *Language Integrator Engine* is in charge of integrating an arbitrary

5. http://pellet.owldl.com

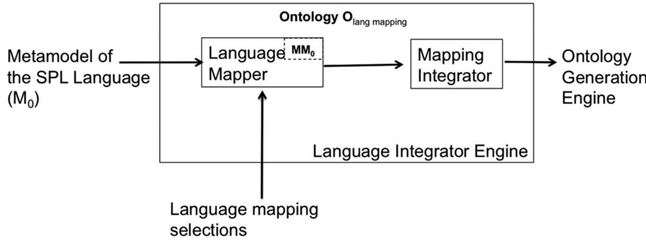

Fig. 4. The Language Integrator Engine. This engine, which is part of Fig. 3a, produces an instance of the Ontology Generation Engine.

SPL language (i.e., an ADL) into OntoAD, as shown in Fig. 4. The integration process basically involves carrying out a mapping of the metamodel M_0 of the SPL language to the meta-metamodel MM_0. The *Language Mapper* is in charge of generating an instance of the ontology $O_{lang\ mapping}$ containing the mapping of the meta-metamodel MM_0 and the metamodel M_0, this based on a set of mapping selections determined by the user. Following, the *Mapping Integrator* produces an instance of the Ontology Generation Engine, previously introduced in Fig. 3a, which is customized to the specific SPL language being integrated. In next sections we outline the meta-metamodel of the OntoAD framework and its integration with our PL-Xelha language.

6.1 The OntoAD Meta-Metamodel

The meta-metamodel of MM_0 is depicted in Fig. 5a. The dotted lines represent optional elements and optional relationships whereas continuous lines denote *Mandatory* elements and relationships. The metamodel defines the architectural *Commonality* and *Variability*. The former involves the SPL language elements, which represent architectural elements that remain fixed along all the members of the product line. There are two kinds of Common elements, namely *Nodes* and *Connections*. A Node involves any SPL language element used to represent fixed architectural elements whereas a Connection represents a binding between such nodes. A Node can optionally have *Ports* functioning as interaction points, hence, Nodes can be connected to each other through Ports. There are *ProvidedPorts* and *RequiredPorts*. The former offer services whereas the latter requires them. Note that Ports are modeled as inner elements of either Nodes or Variant elements.

On the other hand, Variability refers to the SPL language elements representing architectural elements that are variable. More concretely, these elements can be either a *VariationPoint* or a *Variant*. A VariationPoint allows for choosing between different options, which are called Variants. Variants can optionally have Ports. In addition, *VariantConnections* are connections that bind Variants to VariationPoints. In case Variants have Ports, a VariantConnection can bind a Variant's ProvidedPort to a VariationPoint.

The aforementioned elements have a type as an attribute. This attribute is used by the designer to define the name of an architectural element in the SPL language. c_melp_linux is an example of a value for the type attribute of a Variant element. A Variant element additionally includes the *Feature* and *DependsOn* attributes. The *Feature* attribute is used by the designer to specify that feature *Fa* is realized by its associated architectural variant. In the running example, Melp is the value for the attribute *Feature* of the architectural variant c_melp_linux. This means that the designer mapped the feature Melp, which represents an audio compression scheme, to the architectural variant c_melp_linux. The *DependsOn* attribute is employed by the designer to define an include dependency with feature *Fa*. For instance, the designer can state that the selection of the architectural variant c_melp_linux depends on the selection of the feature Linux as this operating systems is required for the instantiation of this architectural variant.

Finally, an element part of the architectural Variability in an SPL is instantiated by (i.e., transformed to) a Node when a particular product architecture *P* is derived from an SPL architecture *A*. For instance, a VariationPoint can be instantiated by a Node. In the case of optional elements, it should be noted that the associations isSource_cn, isTarget_cn, isSource_vcn can be alternatively associated with optional architectural

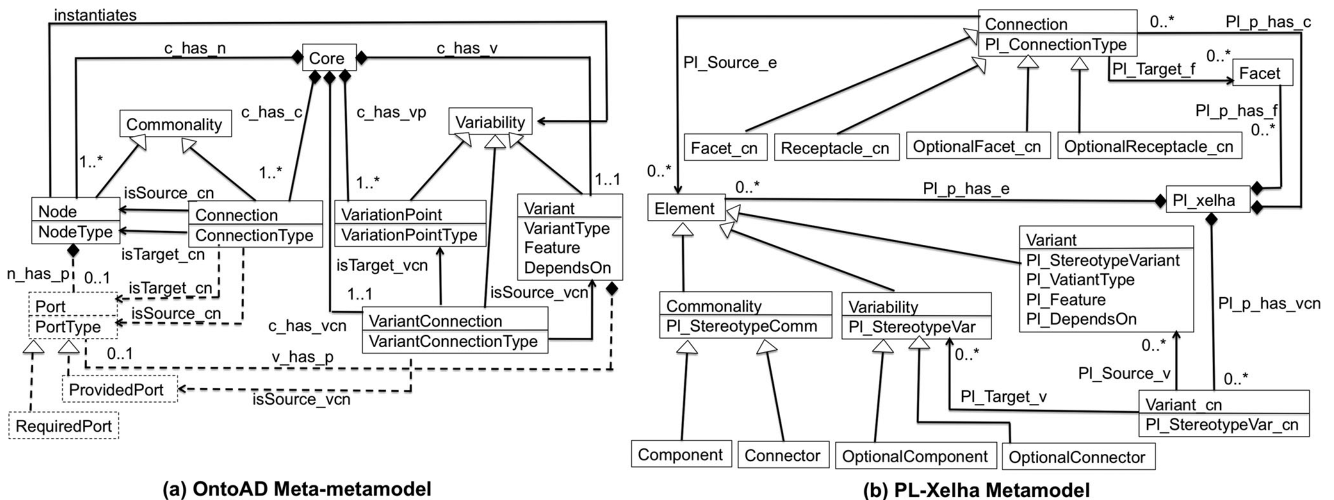

Fig. 5. The Metamodels.

elements. For instance, isSource_cn associates Connection with Node but can alternatively be associated with Port instead.

6.2 The Metamodel of PL-Xelha

Xelha [16] is an ADL that represents the architecture design of distributed multimedia systems. We have developed PL-Xelha, which is an extension of Xelha. An architecture design in PL-Xelha prescribes the commonalities and variabilities of a product line. Fig. 5b shows the metamodel of PL-Xelha and includes Components, which are represented as UML classes stereotyped as <<component>>. Connectors are depicted as UML classes stereotyped as <<connector>> and customized as rectangles. Solid lines denote the commonalities of an architecture while dotted lines denote a variation point, which can be represented either by an OptionalComponent or an OptionalConnector. OptionalComponents and OptionalConnectors representing variation points are stereotyped as <<optionalComponent>> and <<optionalConnector>>, respectively. In addition, Variants are denoted by solid lines and associated with variation points, and are stereotyped as <<variant>>. Finally, we have constructed a visual editor for PL-Xelha in the Eclipse's GMF plugin .[6]

6.3 Language Mapper

The Language Mapper, which is part of the Language Integrator Engine, shown in Fig. 4, offers a graphical user interface whereby the user is able to perform the language mapping. The Language Mapper is implemented in two modules. The first module is developed in Acceleo and is in charge of extracting relevant information from M_0 including the classes, the attributes of the classes, the association relationships of the classes, and the aggregation relationships coming from the root element. This information is given as input to the second module that is implemented in Java. The information of MM_0 such as the classes, the class attributes, and the associations is hard-coded in a Java module. This module presents a graphical interface that allows the user to perform the mapping between MM_0 and M_0 and generates as an output the ontology $O_{lang\ mapping}$.

Each one of the classes Variant and VariantConnection in MM_0 is mapped to a single class in M_0. For example, VariantConnection is mapped to Variant_cn in PL-Xelha. On the other hand, the classes VariationPoint, Node, and Connection in MM_0 can be mapped to one or more classes in M_0. For example, the former is mapped to OptionalComponent and OptionalConnector in PL-Xelha.

Class attributes in the metamodel MM_0 are mapped to the class attributes in M_0. For instance, given that Component is a Node in PL-Xelha, NodeType in MM_0 is mapped to Pl_CommonalityType in M_0.

Target and source association relationships in MM_0 are mapped to target and source association relationships in M_0 altogether with their related Nodes in M_0.

Regarding the instantiates association relationship in MM_0, a Node instantiates a Variability. That is, a Node in the SPL architecture replaces a Variability in the product architecture. For example, a Component instantiates an OptionalComponent whereas a Connector instantiates an OptionalConnector in PL-Xelha. Therefore, the instantiates association relationship is mapped to both OptionalComponent and OptionalConnector.

6. http://www.eclipse.org/modeling/

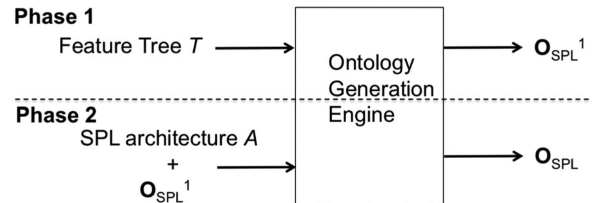

Fig. 6. Model-to-text transformations carried out by the Ontology Generation Engine.

The aggregation relationships c_has_v and c_has_vcn in the meta-metamodel MM_0 are mapped to aggregation relationships in the metamodel M_0. For instance, the relationship c_has_vcn in MM_0 is related to VariantConnection. Hence, in PL-Xelha, c_has_vcn is mapped to Pl_p_has_vcn given that Variant_cn is an instance of MM_0's VariantConnection. In contrast, the aggregation relationships c_has_n, c_has_VP, and c_has_c in MM_0 can be related to multiple elements as indicated by their cardinality. Each of these elements could have a different name for such aggregation relationships in M_0.

6.4 Mapping Integrator

The Mapping Integrator receives as input the ontology $O_{lang\ mapping}$ and yields as output an instance of the Ontology Generation Engine, as shown in Fig. 4. The Mapping Integrator was implemented in Java and produces a program in Acceleo representing the Ontology Generation Engine. The Mapping Integrator performs queries to the ontology $O_{lang\ mapping}$ to obtain the mappings of all elements in MM_0, which are integrated to an instance of the Ontology Generation Engine. Such queries include the following isCommonalityNode some CommonalityNode, isVariationPoint some VariationPoint, and isVariant some Variant, which are used to obtain M_0's common nodes, variation points, and variants, respectively. For instance, in PL-Xelha, the query isVariationPoint some VariationPoint gives the following result: OptionalComponent, and OptionalConnector. In the following sections we present details of the product architecture derivation phase.

7 Ontology Generation Engine

The Ontology Generation Engine, previously introduced in Fig. 3a, is responsible for carrying out a model-to-text transformation in which both a feature tree T and an SPL architecture A are transformed to an instance of the ontology O_{SPL} in Turtle.

The Ontology Generation Engine is tied to a particular meta-model M_0 corresponding to a specific SPL language. However, as mentioned earlier, an instance of the Ontology Generation Engine is automatically generated. The Ontology Generation Engine carries out the model-to-text transformation in two phases, as shown in Fig. 6.

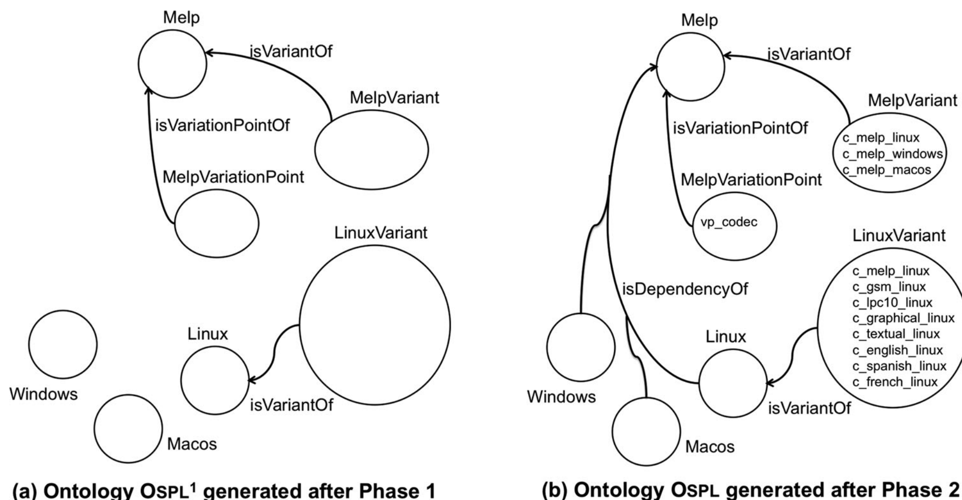

Fig. 7. A subset of the ontologies generated after phase 1 and phase 2 of the Ontology Generation Engine.

7.1 Transformation Rules. Phase 1

In this phase the Ontology Generation Engine transforms a feature tree *T* to the ontology O_{SPL}^{1} and includes the following rules:

Rule 1.1. Transform a feature to an OWL class. The nodes of the feature tree are transformed to OWL classes. The generated classes are placed as subclasses of the OWL class `Features`. For instance, in the running example the feature `Melp` is transformed to an OWL class named `Melp`. A node *N1* that is parent of *N2* in a feature tree *Fi* is transformed to the OWL classes *C1* and *C2*, respectively. The parent relationship between *N1* and *N2* is transformed to the property that states that *C2* is a subclass of *C1*.

Rule 1.2. Transform a variant to an OWL class whose members represent architectural variants. The variants that are leafs in the feature tree are transformed to OWL classes. The generated classes are placed as subclasses of the OWL class `Variant`. For instance, `Melp` is a feature that is both a leaf and a variant. Hence, this feature is transformed to an OWL class named `MelpVariant`, which has associated the following restriction: `isVariantOf some Melp`. It is also generated the property that states that `MelpVariant` is a subclass of the OWL class `Variant`. The members of these OWL classes are defined in the next phase.

Rule 1.3. Transform a variant to an OWL class whose member represents an architectural variation point. The variants that are leafs in the feature tree are transformed to OWL classes. The generated classes are placed as subclasses of the OWL class `VariationPoint`. As an example consider again the feature `Melp`. This feature is transformed to an OWL class named `MelpVariationPoint`, which contains the following existential restriction: `isVariationPointOf some Melp`. It is also generated the property that states that `MelpVariationPoint` is a subclass of the OWL class `VariationPoint`. The members of these OWL classes are defined in the next phase.

Considering our running example, in Fig. 7a, we show a graphical representation of part of the ontology O_{SPL}^{1} that is produced after applying the transformation rules of phase 1. Such ontology only includes OWL classes and some relationships.

7.2 Transformation Rules. Phase 2

In the second phase, the Ontology Generation Engine transforms an SPL architecture *A* to the ontology O_{SPL} by adding information about such architecture to the ontology O_{SPL}^{1}, generated in phase one.

Rule 2.1. Transform architecture dependencies to OWL existential restrictions. Dependencies of architectural variants are transformed to OWL existential restrictions. In the running example, the architectural variant `c_melp_linux` is transformed to a) the following existential restriction, which is added to the `Linux` OWL class: `isDependencyOf some Melp` and b) `c_melp_linux` is included as a member of the OWL class `LinuxVariant` (previously created in phase one). These two transformations take place since the architectural variant `c_melp_linux` has defined the attribute `Feature` equal to feature `Melp` and the attribute `DependsOn` equal to feature `Linux`.

Rule 2.2. Transform architectural variants to members of OWL classes. An architectural variant is transformed to a member of its associated OWL class (which involves its associated feature). In our running example, the following architectural variants are transformed to members of the `MelpVariant` class: `c_melp_linux`, `c_melp_macos`, and `c_melp_windows`. This is so since each one of these architectural variants has defined `Melp` as the attribute `Feature`.

Rule 2.3. Transform architectural variation points to members of OWL classes. An Architectural variation point is transformed to a member of its associated OWL class (which involves its associated feature). In our running example, the `vp_codec` architectural variation point is transformed to a member of the `MelpVariationPoint` class. This is due to the fact that the variation point `vp_codec` is linked up with three variants which have defined `Melp` as the attribute `Feature`.[7]

For our running example, in Fig. 7b, we show a graphical representation of part of the ontology O_{SPL} that is produced after applying the transformation rules of phase 2. In

7. An example of an instance of the Ontology Generator Engine is described in [46].

addition to the OWL classes of the proposed ontology, we also include dependency relationships between classes as well as variation points and variants such as we can see in the classes of the ontology in Fig. 7b.

8 QUERY INTERFACE

The Query Interface module, previously introduced in Fig. 3a, is in charge of producing ontology queries based on feature selections. This module was developed in Java and uses the OWL API library [8] to perform queries on the Ontology-based Reasoning Engine. The Query Interface receives as input the ontology O_{SPL} and the selected features, which are *non-Mandatory* leaves in the feature tree. As a consequence, a set of queries *Q1* are produced. Such queries are applied on the ontology O_{SPL}. The query interface uses the result of the queries to produce a set of tuples *t = {VP, variant}*, each tuple involving an architectural variation point and its selected architectural variant, respectively.

The Query Interface generates three types of queries defined using the Manchester OWL syntax . A query *q1* is used to obtain the architectural variation point associated with a particular feature. *q1* retrieves the OWL class member that has the property of being an architectural variation point related to feature *featurei*: `isVariationPointOf some featurei`. *q2* determines all the dependencies of *featurei* . In other words, *q2* retrieves the set of OWL classes (which in this case represent features) that have the property of being a dependency of feature *featurei*: `isDependencyOf some featurei`. Finally, query *q3*, `isVariantOf some featurei`, retrieves all OWL class members that represent variants of feature *featurei*.

Based on the three types of queries, the algorithm shown below is used to obtain the architectural variant that instantiates a variation point. This association between variants and variation points is represented by tuples of the form *{VP, variant}*. The algorithm also detects mapping errors between features and architecture elements, made by the designer.

```
1..for feature i in ListOfFeatures
2....VP = isVariationPointOf some feature i
3....if VP != null
4......d = (isDependencyOf some feature i )
..............INTERSECT ListOfFeatures
5......if d != null
6........for feature k in d; variants = null
7...........variants = variants +
...............(isVariantOf some feature k )
8........variant = (isVariantOf some feature i)
......................INTERSECT variants
9......else
10........variant = (isVariantOf some feature i)
11.....if variant != null
12........tuples = tuples + {VP, variant}
13.....else
14......raise_ERROR(inconsistency, VP, feature i)
```

8.1 Example

As a means to illustrate the algorithm consider the running example in which part of the ontology produced by the Ontology Generation Engine is shown in Fig. 7b. Also consider that a number of feature selections are carried out including `Melp` as the compression scheme for the codec, and `Linux` as the operating system platform. `ListOfFeatures` is the set the of selected features that are *non-Mandatory* leaves (line 1). In case we want to find the architectural variation point and architectural variant associated with the feature `Melp`, we have the following. First, the architectural variation point is obtained by using the query *q1* `isVariationPointOf some Melp` (line 2) which yields to `vp_codec`. Next, the set of features in which `Melp` has a dependency is obtained by performing the query *q2* (line 4): `isDependencyOf some Melp`, which yields to the set `{Linux, Macos, Windows}`. Then, it is performed an intersection of the former set and the set of selected features which happens to be `{Melp, Graphical, English, Spanish, One2one, Linux, Recording}`, which yields to `d = {Linux}` (line 4). Then the architectural variants associated with feature `Linux` are found by performing the query *q3* `is VariantOf some Linux` (line 7). Such a query gives as a result the following architectural variants:

`c_graphical_linux`, `c_french_linux`,
`c_gsm_linux`, `c_lpc10_linux`,
`c_melp_linux`, `c_spanish_linux`,
`c_textual_linux`, `c_english_linux`.

Hence, the query to obtain the architectural variant associated with feature `Melp` is formulated with the following query involving an intersection (line 8):

`isVariantOf some Melp and isVariantOf some Linux`

The result of this query is `c_melp_linux`.[9] Thus, for the selected feature `Melp` the Ontology-based Reasoning Engine produces the following output: `{vp_codec, c_melp_linux}`. The element `c_melp_linux` is the architectural variant that instantiates the architectural variation point `vp_codec`.

As a means to illustrate the detection of inconsistencies in the SPL architecture specification, consider that the following constraint is wrongly defined by the designer[10]: `c_melp_linux, Feature = Melp, DepensOn = Windows`. Meaning that the feature `Melp` is mapped to the component `c_melp_linux` and that this component has a dependency with the feature `Windows` (this wrongly specified as the dependency should be stated with the feature `Linux` instead). Similarly as stated before, the query to obtain the architectural variant associated with feature `Melp` is formulated with the following query involving an intersection (line 8): `isVariantOf some Melp and isVariantOf some Linux`.

The result of this query is null, thus, raising an inconsistency error (line 14). The reason of this is as follows. The

8. http://sourceforge.net/projects/owlapi/

9. It should be noted that this query is equivalent to the following query: `isVariantOf some Melp INTERSEC variants` where `variants = {c_graphical_linux, c_french_linux, c_gsm_linux, c_lpc10_linux, c_melp_linux, c_spanish_linux, c_textual_linux, c_english_linux}`. The query `isVariantOf some Melp` yields to the set `{c_melp_linux, c_melp_macos, c_melp_windows}`, therefore the result of the intersection is `c_melp_linux`.

10. Note that this constraint can be expressed with Boolean formulas as follows: `c_melp_linux requires Melp and Windows`.

query `isVariantOf` some `Melp` yields to the set {`c_melp_linux, c_melp_macos, c_melp_windows`}, however the query `isVariantOf` some `Linux` yields to the set {`c_graphical_linux, c_french_linux, c_gsm_linux, c_lpc10_linux, c_spanish_linux, c_textual_linux, c_english_linux`}. The intersection of the two sets is null. It should be noted that the latter set includes all the architectural elements that have a dependency with feature `Linux`. As a consequence, `c_melp_linux` is not part of this set as we considered this dependency was wrongly specified, by the designer, with the feature `Windows`.

Our algorithm is also able to detect when the designer misses to map a feature to an architecture element. For instance, in case the feature `Recording` is not mapped to any architecture element, the query to obtain the architectural variant associated with feature `Recording` is formulated with the following query (line 8): `isVariantOf` some `Recording`. The result of this query is null, hence, also raising an inconsistency error (line 14).

9 Rule Generator Engine

The Rule Generator Engine, previously introduced in Fig. 3a, is in charge of producing ATL rules [33]. The Rule Generator Engine receives as input a set of tuples *t*, which are generated by the Query Interface. As said earlier, the first element of a tuple is an architectural variation point and the second element is the associated architectural variant that will instantiate such a variation point. The Rule Generator Engine receives also as input the ontology $O_{lang\ mapping}$. The knowledge contained in $O_{lang\ mapping}$ is used to achieve independence of the SPL language. A set of queries *Q2* are employed on $O_{lang\ mapping}$ to aid the generation of such rules. Once the rules are produced, the Rule Generator Engine invokes an ATL transformation engine to obtain the transformation from the SPL model to the product model, whereby the derivation of the product architecture is obtained.

9.1 An Algorithm for Generating M2M Transformation Rules

For the sake of brevity we focus only on presenting the process to generate model-to-model transformation rules. The algorithm shown below is used to produce such rules. The first part of the algorithm generates rules that transforms an instance of *MM0's* `VariationPoint` to an instance of *MM0's* `Node`. The algorithm involves queries made to the ontology $O_{lang\ mapping}$.

1...// transform variation points to nodes
2...for **commonalityNode_M** in
.....isCommonalityNode some CommonalityNode
3........if(isInstantiatedBy some commonalityNode_M != NULL)
4.............**variability_M** =
.................................isInstantiatedBy some commonalityNode_M
5.............**varType_M** =
.................................isElementTypeFieldOf some variability_M
6.............**commoType_M** =
........................isElementTypeFieldOf some commonalityNode_M
7.............for {**variationPoint_A**, **variant_A**} in t
8.........................generate transformation rule

First, the algorithm queries the ontology $O_{lang\ mapping}$ the common elements in the SPL language (line 2). A transformation rule is produced only for those common nodes that are able to instantiate a variable element (line 3). For example, `Component`, `Connector` and `Facet` are common elements in PL-Xelha. However, only the two former instantiate a variable element. Next, it is queried the variable element that each common node instantiates (line 4). For instance, the common node `Component` instantiates the variable element `OptionalComponent` in PL-Xelha. In other words, an `OptionalComponent` in *A* is transformed to a `Component` in *P*. Also, it is obtained the variable's element type (line 5) as well as the common node's element type (line 6). For each tuple {`variationPoint_A, variant_A`}, a transformation rule is generated (lines 7-8).

9.2 Transformation Rule Template

Given that *M0* is the metamodel of an SPL language *L* and *A* a specific SPL architecture defined in *L*, our algorithm uses the template shown below to generate a transformation rule.

rule **variability_M2commonalityNode_M** {
........from
.................s : splLang!**variability_M** (
...............................s.**VarType_M** = '**variationPoint_A**')
.........to
.................t : productLang!**commonalityNode_M**
.................do {
...........................if(s.**varType_M** = '**variationPoint_A**') {
...................................t.**commType_M** <- '**variant_A**'; }
..................}
}

This is achieved by replacing the text in bold by values of variables that are specific to an SPL language and an SPL architecture whereas the rest of the text remains fixed for all rules. Such variables are the following:

a) `variability_M` is an *M0's* variation point (e.g. `OptionalComponent` in PL-Xelha).
b) `commonalityNode_M` is an *M0's* common node (e.g. `Component` in PL-Xelha)
c) `commType_M` is an *M0's* common type (e.g. `Pl_CommonalityType` in PL-Xelha)
d) `varType_M` is an *M0's* variability type (e.g. `Pl_VariabilityType` in PL-Xelha)
e) `variant_A` is an *A's* variant (e.g. `c_melp_linux`)
f) `variantionPoint_A` is an *A's* variation point (e.g. `vp_codec`)

The following ATL code is an example of an `OptionalComponent` to `Component` transformation rule for PL-Xelha whereby `vp_recording` is transformed to `c_recording`, which is illustrated in Fig. 8. This transformation is optional, meaning that `vp_recording` suffers a transformation only in case the *Optional* feature `Recording` is selected. The ATL code also defines the transformation of `vp_codec` to `c_melp_linux` as shown in Fig. 9. `vp_codec` represents the *Alternative* variation point `Compression Scheme` of our feature tree which

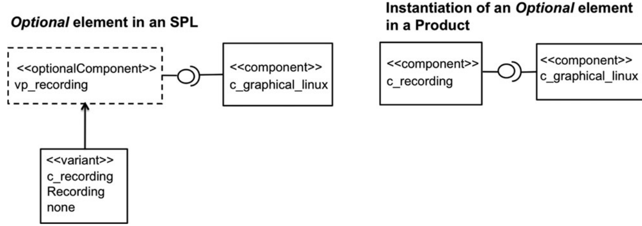

Fig. 8. Transformation of an *Optional* element in PL-Xelha.

has Melp, Gsm, and Lpc10 as features. Hence, being vp_codec an *Alternative* variation point, the Rule Generator Engine must always generate a rule that transform vp_codec to a particular component to ensure the derived product is consistent.

```
rule OptionalComponent2Component {
................from
........s : splLang!OptionalComponent (
................s.Pl_VariabilityType = 'vp_codec' or
................s.Pl_VariabilityType = 'vp_recording'
........to
................t : productLang!Component
..............do {
.....................if(s.Pl_VariabilityType = 'vp_codec')
.............................{ t.Pl_CommonalityType <- 'c_melp_linux'; }
.....................if(s.Pl_VariabilityType = 'vp_recording')
.............................{ t.Pl_CommonalityType <- 'c_recording'; }
...............}
}
```

The ATL code of an OptionalConnector to Connector transformation rule for PL-Xelha is very similar (further details can be found in [46]).

In case we select the features Melp, Linux, Graphical, English, Spanish, One2one, and Recording, the Rule Generator Engine produces rules that transforms the VoIP SPL architecture to the specific product shown in Fig. 10. Given that the architecture of the sender and receiver are symmetrical, we only show the sender part of the system.

10 Evaluation

We present a performance evaluation and test the scalability of our approach. We also present an evaluation of other aspects of our framework such as mapping error detection, product architecture derivation, the compactness of the derivations, and language independence.

10.1 Performance Evaluation

In this section we report the results of the evaluation using the proposed framework and transformation rules, and test the scalability of our approach. Given that we are using an ontology-based reasoning engine, it is relevant this kind of evaluation since it has been shown elsewhere [47], [48], [49], [50], [51] that such reasoning engines present problems of scalability for large ontologies.

In our approach, the size of the ontologies employed is proportional to the size of the feature tree involved. Therefore, feature trees involving hundreds of features have associated large ontologies in our framework.

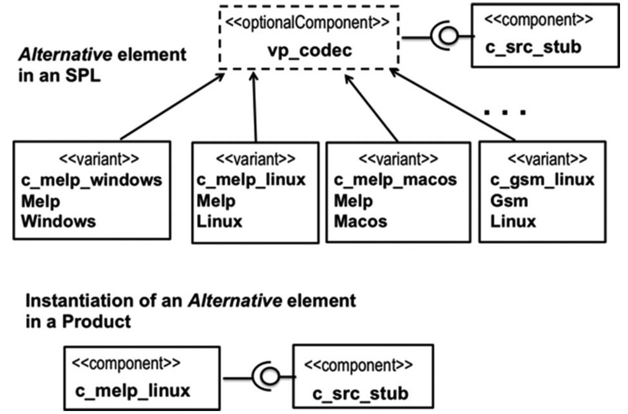

Fig. 9. Transformation of an *Alternative* element in PL-Xelha.

10.1.1 Experimental Setup

We carried out our experiment on a MacBook Pro dual core Intel Core i5 at 2.5 GHz with 3 MB of L3 cache memory and 4 GB of RAM running Mac OS X, version 10.8. The performance of the prototype was evaluated for different sizes of feature trees and SPL architectures. In our experiment we tested the scalability of the following modules: Ontology Generation Engine, Query Interface, and Rule Generator Engine. The Ontology-based Reasoning Engine was indirectly tested by the experiments carried out for both the Query Interface and the Rule Generator Engine, as these modules perform queries over the ontologies. The performance of the Model-to-model Transformation Engine consisting of an ATL engine has been tested elsewhere [52] and it has been found that the performance degradation of an ATL engine is linear.

Our experimental evaluation focuses on a proof of concept. In order to carry out the experiments we developed two Java programs, called *Testbed1* and *Testbed2*. The former generates features trees including *Mandatory*, *Optional*, *Alternative*, and *Or* relationships with different sizes whose feature names are random. A feature tree generated is represented in an XML-based file readable by both the feature editor tool [45] and the Ontology Generation Engine. *Testbed1* also generates a random selection of features. In addition, *Testbed2* produces random SPL architectures, specified in PL-Xelha, whose architectural variants are mapped to the features contained in the feature trees generated.[11] *Testbed2* creates an architectural variation point for each feature that is a variation point.[12] Also, one architectural variant is mapped to a feature tree's variant. The SPL architectures generated are represented in an XML-based file readable by both the PL-Xelha editor and the Ontology Generation Engine.

In the experimental settings, the number of architectural variation points corresponds to 30 percent of the number of features. Moreover, the number of architectural variation points is 10 percent of the architectural variants. Also, 90 percent of the total amount of architectural variants have

11. This is achieved by populating the Feature attribute of the architectural variants with the feature names.

12. A feature tree's variation point is a feature, in which at least one of its child is a non-mandatory feature leave.

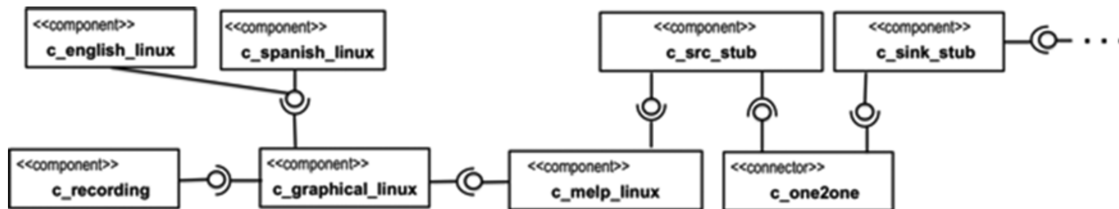

Fig. 10. A specific product of the VoIP system in PL-Xelha.

dependencies on three features. [13] We decided to take these proportions as these were the proportions shown by our VoIP running example. Our prototype was tested for 1,000, 2,000, 3,000, 4,000, and 5,000 features, which were associated with different sizes of an SPL architecture, as shown in Table 1. For instance, a tree of 1,000 features has associated an SPL architecture with 300 architectural variation points and 3,000 architectural variants.

All experimental runs were performed five times, the average of these runs was considered. In all experimental scenarios the sampling distribution was nearly normal, hence, based on the central limit theorem we considered the sample size (i.e., five runs) to be large enough. The maximum relative standard deviation obtained was 20, 11, and 2 percent for the first, second, and third experimental scenarios, respectively. Each repetition of an experimental run involved a feature tree of the same size in terms of number of features but with different width, depth, and feature selections. For example, in the first experimental scenario, the experiment for 5,000 features was repeated five times, generating five different feature trees and a different amount of feature selections, as shown in Table 2. The maximum children per node, shown in Table 2, was the same for all experiments,[14] however, the number of feature selections and number of levels changed accordingly to the size of the tree.

10.1.2 Experimental Results

The experimental Scenario 1 tested the scalability of the Ontology Generation Engine for different sizes of both a feature tree and an SPL architecture. The performance of the Ontology Generation Engine was tested for 1,000, 2,000, 3,000, 4,000, and 5,000 features, as shown in Fig. 11. We can observe that the behavior is almost linear. Even in the case of 5,000 features the execution time was only about 7 minutes. Therefore, the Ontology Generation Engine scales well.

The experimental Scenario 2 tested the scalability of the Query Interface for different sizes of the ontology *OSPL*, determined by both the amount of features and architectural variation points. Such ontologies were generated by the experimental Scenario 1, in which each of the five runs for each experiment had associated a different feature tree (i.e., same size but different width and depth) and different feature selections. The execution time of the queries performed by the Query Interface was measured for different sizes of the ontology. Fig. 12 shows the Query Interface scales well and the maximum execution time was of only about 8 minutes for 5,000 features.

The experimental Scenario 3 tested the scalability of the Rule Generator Engine. Each of the five runs for each experiment receives as input the output generated in the experimental Scenario 2, i.e., a set of tuples. Each tuple represents an architectural variation point and its selected architectural variant. We measured the execution time taken to obtain the transformation rules by the Rule Generator Engine for 300, 600, 900, 1,200, and 1,500 architectural variation points. Fig. 13 shows that the execution time required to generate both copy rules and transformation rules is of only 3.5 seconds in the largest case (i.e. 1,500 architectural variation points).

10.2 Product Derivation Evaluation

We tested the derivation of a product architecture for 30 random possible feature configurations with our running example. The number of product derivations was based on the central limit theorem in order to converge to a normal central distribution. In all cases the product architecture was correctly derived. Furthermore, we also tested the detection of mapping errors. First, for each of the 24 architectural variants with dependencies to other features, we tested the mappings of wrong feature dependencies. Second, we tested if any of the 14 features described in the VoIP system were missing during the mappings between these features and architectural elements. In all cases, our prototype detected the mapping error.

In addition, we also compared the compactness of a product architecture derived with a Boolean-based approach and with our approach. The result was that, although both products were semantically equivalent, the product derived with Boolean formulas (without the aid of MDD techniques) was much more complex including twice as more components than that obtained with our approach. Therefore, the former way of product derivation in ADLs

TABLE 1
Relationship between the Size of a Feature Tree and the Size of an SPL Architecture

Features	Architectural VPs	Architectural Variants
1,000	300	3,000
2,000	600	6,000
3,000	900	9,000
4,000	1,200	12,000
5,000	1,500	15,000

13. For instance, in the running example, the feature variant `Melp` has dependencies with features `Linux`, `Macos` and `Windows`. As a result, `Melp` is associated with three architectural variants, namely `c_melp_linux`, `c_melp_macos`, `c_melp_windows`.

14. That is, these values did not change for trees of 1, 2, 3, 4 or 5 thousand features.

TABLE 2
Five different Trees, Each Containing 5,000 Features

Experimental repetition	Maximum Children per Node	Tree Levels	Feature Selections
1	3	25	938
2	4	10	833
3	5	7	895
4	6	6	876
5	7	5	848

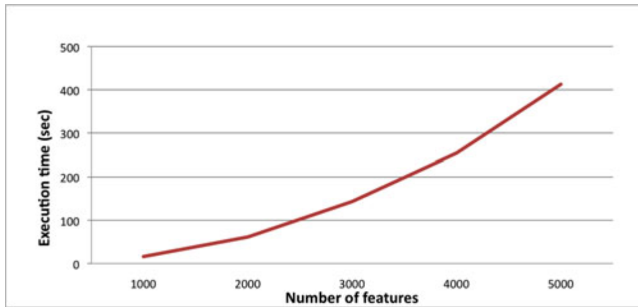

Fig. 11. Scenario 1, performance of the Ontology Generation Engine.

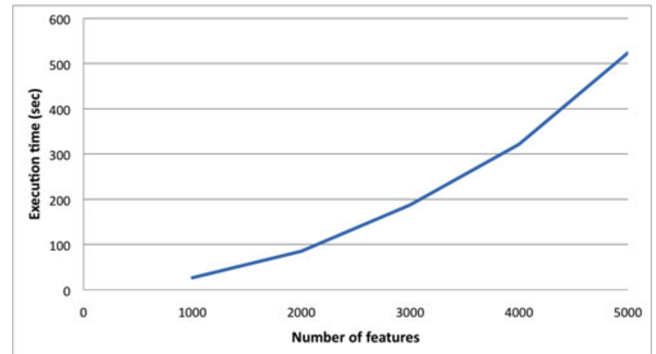

Fig. 12. Scenario 2, performance of the Query Interface.

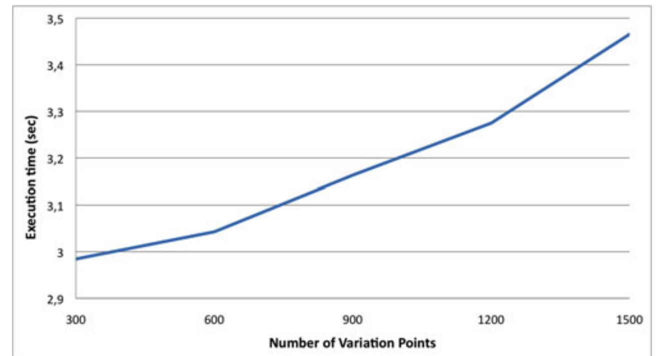

Fig. 13. Scenario 3, performance of the Rule Generator Engine.

can result in producing unnecessary verbose and complex specifications of product architectures, which can be difficult to read and understand. Further details of this comparison can be found in [46].

Finally, our approach only requires the designer has feature modeling and software architecture design skills whereas other approaches require to possess knowledge in at least one further area of expertise such as propositional logic, model-driven development, or aspect oriented programming.

10.3 Discussion

Throughout the running example we have shown the feasibility of the OntoAD framework to automate the derivation of product architectures, as we also demonstrated the language-independence of our framework using two different SPL languages. For an SPL language to be integrated to OntoAD, the meta-model of such a language must conform to the meta-metamodel MM_0 defined as part of the framework. This might be seen as a limitation of our approach since there might be different SPL languages that do not conform to MM_0.[15] However, such languages could be incorporated to OntoAD by extending MM_0 as no restriction is imposed in that respect. A limitation of our approach is that the meta-metamodel MM_0 does not include the property of composite components. Another limitation of our framework is that the product derivation process does not support structural changes (e.g. component interconnection changes) to satisfy quality attributes. These issues involve future work. Currently our approach only considers one-to-one relationships between features and architectural variants since this simplifies the algorithm of the Query Interface. As a result, the designer is forced to insert a single architectural variant to directly support a selected feature. This, without giving the flexibility of using multiple architectural variants if considered convenient by the designer. Future work will include support for one-to-many relationships.

The experimental results have shown that the approach is suitable even for large-scale feature systems. The random SPL architectures generated follow a pipe-and-filter architecture style. The reason of this was simplifying the implementation of *Testbed2* as neither the architecture style nor the number of component interconnections have an impact on the performance of the following modules: Ontology Generation Engine, the Query Interface, and the Rule Generator Engine. The performance of such modules is only impacted by the number of both the architectural variation points and architectural variants that are contained in an SPL architecture. This is due to the fact that the Acceleo program implementing the Ontology Generation Engine only parses architectural variation points and architectural variants, and the rest of the elements in an SPL architecture are ignored. Similarly, only the SPL architecture variation points and number of variants have an impact on the size of the ontology O_{SPL} that the Query Interface receives as input. In addition, the performance of the Rule Generator Engine is only impacted by the number of tuples (received as input) that is determined by the number of SPL architecture's variation points.

In Scenario 1, we can observe that the performance degradation of the Ontology Generation Engine is almost linear. In the largest case tested (i.e. 5,000 features) the execution time took only about 7 minutes. In this scenario, the maximum relative standard deviation was of 20 percent, which

15. Given that the meta-metamodel defines the elements that a metamodel is able to use, a metamodel does not conforms to a meta-metamodel when the former contains either elements or relationships among the elements that are not part of the latter.

could be considered a bit high. We obtained such a deviation value since the performance of the Ontology Generation Engine is impacted by the depth of the tree. That is, the Ontology Generation Engine's performance decreases as more levels has a feature model. This occurs since the Acceleo program, implementing such an engine, employs a recursive algorithm that is negatively impacted by the depth of the feature model. For instance, in the case of feature models encompassing 5,000 features, the number of levels varied between 5 and 25, such as Table 2 shows.

Regarding Scenario 2, we observed that the execution time of the Query Interface took about 8 minutes in the case of 5,000 features. The Query Interface's performance decreases as more ontology queries are performed. The algorithm of the Query Interface employs three ontology queries per selected feature. Hence, in the case of 5,000 features there are around 900 selected features giving a total of 2,700 ontology queries. Also, we obtained a maximum relative standard deviation of 11 percent. The reason such a deviation is a bit high is the Query Interface module is impacted by the number of feature selections and Scenario 2 involved a varying number of feature selections for each experimental run. For example, in case of trees with 5,000 features, the number of feature selections ranged from 833 to 938 selections, as shown in Table 2. Last, the performance overhead produced by the Rule Generator Engine in Scenario 3 is negligible as only 3.5 seconds was required in the largest case tested and the maximum relative standard deviation obtained was of only 2 percent.

The overall performance of OntoAD showed to be almost linear. Our approach took about 15 minutes to derive a product in the largest case. Although, the derivation process of propositional logic-based approaches is usually cheaper, the performance we loose is compensated by the fact that we gain more power of expressiveness and reduce human intervention.

Finally, in order to test language independence, we used Koala [53]. Koala was designed to model variability in an SPL for the consumer electronics industry. Koala is also an ADL as it supports the description of the structure of a configuration in terms of its components. We integrated a customized version of Koala with our framework. Then, we successfully replicated the running example in Koala.[16]

11 Conclusions

In this research we have presented our OntoAD framework, a language independent approach for deriving product-specific architectures from a a generic SPL architecture. Our model-driven approach enables the automatic generation of transformation rules, we use in the derivation of product architectures and an Ontology-based Reasoning Engine that reduces the overhead of manual derivation tasks. Moreover, our models supports different architecture languages as it diminishes the dependency to a particular notation. We also illustrate our approach and the solution adopted via a motivating VoIP example that proves the validity of the approach and performance results.

16. Details of the integration of Koala into our framework can be found in [46].

Our solution enables the identification of mapping errors before transformations of the derivation process happen as this reduces the likelihood of errors introduced manually by the product line engineer. Therefore, our solution ensures a better quality of the derivation process. However, we do not validate other types of errors such as invalid connections between the components in the derived architecture.

The performance results show that our approach works fine to a certain extend depending on the number of features, variation points and variants. We can state that above 4,000-5,000 features the performance of the ontology generation engine and query interface modules decay. In addition, the performance of the rule generation engine decays when the number of variation points is between 1,200-1,500 or greater. However, additional measures and performance tests are necessary to prove better the scalability of our model.

Future work will focus on the verification of the structural compliance of the derived product architecture versus the SPL architecture (e.g. check component interconnections in the derived product are correct) and also verify that the instantiated variants in the resultant architecture conform to the feature selections. Finally, we also suggest to explore how to incorporate one-to-many relationships between features and architectural elements as well as looking into the issue to support the design of composite components in an SPL architecture.

The role of quality attributes and how these may affect to structural changes in the product architectures during the derivation process is another challenging area worthy to explore. All in all, we believe our approach is a first step to enhance some aspects of the transformations commonly used in the mappings between feature models and architecture models as it partially automates a previous step in the derivation process from a generic SPL architecture.

Acknowledgments

The authors are thankful for the valuable feedback and insightful comments of Roberto E. Lopez-Herrejon, which considerably improved the content of this paper.

References

[1] C. W. Krueger, "New methods in software product line practice," *Commun. ACM*, vol. 49, pp. 37–40, Dec. 2006.
[2] D. M. Weiss, P. C. Clements, K. Kang, and C. W. Krueger, "Software product line hall of fame," in *Proc. 10th Int. Softw. Product Line Conf.*, 2006, p. 237.
[3] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Boston, MA, USA: Addison-Wesley, 1998.
[4] D. Batory, "Feature models, grammars, and propositional formulas," in *Proc. Int. Softw. Product Line Conf.*, 2005, pp. 7–20.
[5] D. Benavides, S. Segura, and A. R. Cortés, "Automated analysis of feature models 20 years later: A literature review," *Inf. Syst.*, vol. 35, no. 6, pp. 615–636, 2010.
[6] J. White, D. C. Schmidt, D. Benavides, P. Trinidad, and A. R. Cortés, "Automated diagnosis of product-line configuration errors in feature models," in *Proc. 12th Int. Softw. Product Line Conf.*, 2008, pp. 225–234.
[7] K. Czarnecki and M. Antkiewicz, "Mapping features to models: A template approach based on superimposed variants," in *Proc. 4th Int. Conf. Generative Programm. Component Eng.*, 2005, vol. 3676, pp. 422–437.

[8] E. M. Dashofy, A. van der Hoek, and R. N. Taylor, "A comprehensive approach for the development of modular software architecture description languages," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 2, pp. 199–245, 2005.

[9] T. Asikainen, T. Männistö, and T. Soininen, "Kumbang: A domain ontology for modelling variability in software product families," *Adv. Eng. Informat.*, vol. 21, no. 1, pp. 23–40, 2007.

[10] D. Perovich, P. O. Rossel, and M. C. Bastarrica, "Feature model to product architectures: Applying mde to software product lines," in *Proc. Joint Working IEEE/IFIP Conf. Softw. Archit. Eur. Conf. Softw. Archit.*, 2009, pp. 201–210.

[11] G. Botterweck, L. O'Brien, and S. Thiel, "Model-driven derivation of product architectures," in *Proc. 22nd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2007, pp. 469–472.

[12] C. Parra, A. Cleve, X. Blanc, and L. Duchien. (2010). Feature-based composition of software architectures in *Proc. 4th Eur. Conf. Softw. Archit.*, pp. 230–245. [Online]. Available: http://hal.inria.fr/inria-00512716/en/

[13] F. Heindenreich, "Towards systematic ensuring well-formedness of software product lines," in *Proc. 1st Int. Workshop Feature-Oriented Softw. Develop.*, 2009, pp. 69–74.

[14] B. Selic, "Model-driven development: Its essence and opportunities," in *Proc. 9th IEEE Int. Symp. Object Component-Oriented Real-Time Distrib. Comput.*, 2006, pp. 313–319.

[15] A. Spanias, *Speech Coding Standards*, G. Gibson, Ed. New York, NY, USA: Academic, 2000.

[16] H. A. Duran-Limon and G. S. Blair, "QoS management specification support for multimedia middleware," *J. Syst. Softw.*, vol. 72, no. 1, pp. 1–23, 2004.

[17] S. Apel, C. Kastner, and C. Lengauer, "Language-independent and automated software composition: The featurehouse experience," *IEEE Trans. Softw. Eng.*, vol. 39, no. 1, pp. 63–79, Jan. 2013.

[18] D. Batory, J. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *IEEE Trans. Softw. Eng.*, vol. 30, no. 6, pp. 355–371, Jun. 2004.

[19] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann, "An overview of caesarj," in *Transaction Aspect-Oriented Software Development I*, A. Rashid and M. Aksit, Eds. Berlin Heidelberg, Springer, 2006, vol. 3880, pp. 135–173.

[20] A. Bergel, S. Ducasse, and O. Nierstrasz, "Classbox/j: Controlling the scope of change in Java," in *Proc. 20th Annu. ACM SIGPLAN Conf. Object-Oriented Programm., Syst., Lang. Appl.*, 2005, pp. 177–189.

[21] S. Apel, T. Leich, M. Rosenmller, and G. Saake, "Featurec++: On the symbiosis of feature-oriented and aspect-oriented programming," in *Generative Programming and Component Engineering*, R. Glck and M. Lowry, Eds. Berlin, Heidelberg, Springer, 2005, vol. 3676, pp. 125–140.

[22] S. Apel, S. Kolesnikov, J. Liebig, C. Kstner, M. Kuhlemann, and T. Leich, "Access control in feature-oriented programming," *Sci. Comput. Programm.*, vol. 77, no. 3, pp. 174–187, 2012.

[23] K. Bak, K. Czarnecki, and A. Wasowski, "Feature and metamodels in clafer: Mixed, specialized, and coupled," in *Proc. 3rd Int. Conf. Softw. Lang. Eng.*, 2010, pp. 102–122.

[24] M. Pinto, L. Fuentes, and L. Fernndez. (2012). Deriving detailed design models from an aspect-oriented {ADL} using {MDD}, *J. Syst. Softw.* [Online]. *85(3)*, pp. 525–545. Available: http://www.sciencedirect.com/science/article/pii/S0164121211001269

[25] M. Asadi, E. Bagheri, D. Gasevic, and M. Hatala, "Goal-oriented requirements and feature modeling for software product line engineering," in *Proc. GPCE*, 2010, http://www.ee.ryerson.ca/~bagheri/papers/gpce10_1.pdf

[26] Ø. Haugen, "Common Variability Language (CVL) – revised submission," in *OMG document ad/2012-08-05*, OMG 2012.

[27] K. Czarnecki, C. H. P. Kim, and K. T. Kalleberg, "Feature models are views on ontologies," in *Proc. 10th Int. Softw. Product Line Conf.*, 2006, pp. 41–51.

[28] W. Peng, Xim an Zhao, Y. Xue, and Y. Wu, "Ontology-based feature modeling and application-oriented tailoring," in *Proc. 9th Int. Conf. Reuse Off-the-Shelf Components*, 2006, pp. 87–100.

[29] L. A. Zaid, F. Kleinermann, and O. De Troyer, "Applying semantic web technology to feature modeling," in *Proc. ACM Symp. Appl. Comput.*, 2009, pp. 1252–1256.

[30] M. F. Johansen, F. Fleurey, M. Acher, P. Collet, and P. Lahire, "Exploring the synergies between feature models and ontologies," in *Proc. SPLC Workshops*, 2010, pp. 163–170.

[31] C. Quinton, N. Haderer, R. Rouvoy, and L. Duchien, "Towards multi-cloud configurations using feature models and ontologies," in *Proc. Int. workshop Multi-Cloud Appl. Federated Clouds*, 2013, pp. 21–26.

[32] A. Murguzur, R. Capilla, S. Trujillo, Ó. Ortiz, and R. Lpez-Herrejn, "Context variability modeling for runtime configuration of service-based dynamic software product lines," in *Proc. 18th Int. Softw. Product Line Conf.: Companion Volume Workshops, Demonstrations Tools*, 2014, pp. 2–9.

[33] J. Bzivin, G. Dup, F. Jouault, G. Pitette, and J. E. Rougui, "First experiments with the ATL model transformation language: Transforming XSLT into xquery," in *Proc. 2nd OOPSLA Workshop Generative Techn. Context Model Driven Archit.*, 2003, http://s23m.com/oopsla2003/bezivin.pdf

[34] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch, "Covamof: A framework for modeling variability in software product families," in *Proc. 3rd Int. Softw. Product Line Conf.*, 2004, pp. 197–213.

[35] J. Perez, J. Diaz, C. Costa-Soria, and J. Garbajosa, "Plastic partial components: A solution to support variability in architectural components," in *Proc. Joint Working IEEE/IFIP Conf. Softw. Archit. Eur. Conf. Softw. Archit.*, Sep. 2009, pp. 221 –230.

[36] M. Razavian and R. Khosravi, "Modeling variability in the component and connector view of architecture using uml," in *Proc. ACS/IEEE Int. Conf. Comput. Syst. Appl.*, 2008, vol. 0, pp. 801–809.

[37] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Trans. Softw. Eng.*, vol. 26, no. 1, pp. 70–93, Jan. 2000.

[38] M. Horridge, N. Drummond, S. Jupp, G. Moulton, and R. Stevens. (2009, Mar.). A practical guide to building owl ontologies using the protege-owl plugin and co-ode tools edition 1.2. [Online]. Available: http://www.co-ode.org/resources/tutorials/ProtegeOWLTutorial.pdf

[39] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, *The Description Logic Handbook*. New York, NY, USA: Cambridge Univ. Press, 2007.

[40] W3c, turtle. [Online]. Available: http://www.w3.org/TeamSubmission/turtle/, 2011.

[41] C. Gonzalez-Perez and B. Henderson-Sellers, *Metamodelling for Software Engineering*, Hoboken, NJ, USA: Wiley, 2008.

[42] Acceleo. [Online]. Available: http://www.acceleo.org/, 2015.

[43] O. Pastor and J. C. Molina, *Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling*, 1st ed. New York, NY, USA: Springer, 2010.

[44] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*, 2nd ed. Reading, MA, USA: Addison-Wesley, 2008.

[45] A. Gomez and I. Ramos, "Cardinality-based feature modeling and model-driven engineering: Fitting them together," in *Proc. 4th Int. Workshop Variability Modelling Softw.-Intensive Syst.*, 2010, pp. 61–68.

[46] H. A. Duran-Limon, C. A. Garcia-Rios, E. Castillo-Barrera, and R. Capilla. (2015). An ontology-based product architecture derivation approach: A complementary document. [Online]. Available: http://maestro.cucea.udg.mx/ hduran/publications/OntoAD-comp.pdf

[47] J. Bock, P. Haase, Q. Ji, and R. Volz, "Benchmarking owl reasoners," in *Proc. ARea2008 Workshop*, 2008, http://ceur-ws.org/Vol-350/paper1.pdf

[48] S. Abburu, "A survey on ontology reasoners and comparison," *Int. J. Comput. Appl.*, vol. 57, no. 17, p. 33, 2012.

[49] P. Hitzler and D. Vrandecic, "The screech owl reasoner-scalable approximate abox reasoning with owl," in *Proc. Int. Semantic Web Conf., Softw. Demo*, 2005.

[50] P. Hitzler, M. Krötzsch, B. Parsia, P. F. Patel-Schneider, and S. Rudolph, "Owl 2 web ontology language primer," *W3C Recommendation*, vol. 27, no. 1, p. 123, 2009.

[51] L. Lefort, K. Taylor, and D. Ratcliffe, "Towards scalable ontology engineering patterns: Lessons learned from an experiment based on w3c's part-whole guidelines," in *Proc. 2nd Australasian Workshop Adv. Ontol.-Volume 72*, Darlinghurst, Australia, 2006, pp. 31–40.

[52] M. Amstel, S. Bosems, I. Kurtev, and L. Ferreira Pires, "Performance in model transformations: Experiments with atl and qvt," in *Theory and Practice of Model Transformations*, vol. 6707, J. Cabot and E. Visser, Eds. Berlin, Heidelberg, Springer, 2011, pp. 198–212.

[53] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The koala component model for consumer electronics software," *IEEE Comput.*, vol. 33, no. 3, pp. 78–85, Mar. 2000.

Hector A. Duran-Limon received the PhD degree at Lancaster University, England in 2002. He is currently a full professor at the Information Systems Department, University of Guadalajara, Mexico. Following this, he was a post-doctoral researcher until December 2003. He received IBM Faculty award in 2008. His research interests include cloud computing and high-performance computing (HPC). He is also interested in software architectures, software product lines, and component-based development. In 2006, he was invited to create a PhD program in information technologies for the University of Guadalajara, becoming a member of the academic-council. He is a member of the IEEE.

Carlos A. Garcia-Rios received the bachelor's degree in computing engineering at the University of Guadalajara, Mexico. His current interests involve software architectures, software product lines, and component-based development.

Francisco E. Castillo-Barrera received the master's degree in computer science from the Institute of Research in Applied Mathematics and Systems (IIMAS), the National Autonomous University of Mexico and received the PhD degree at University of Guadalajara, Mexico in 2012. He works as a full-time professor and researcher at the University of San Luis Potosi, Mexico. His research interests are: Artificial intelligence, software engineering, semantic web, ontologies, artificial life, virtual animated agents based on logic, knowledge representation, software components, logic programming. He is a member of the IEEE

Rafael Capilla received the PhD degree in computer science in 2004 and is an associate professor in the Computer Science Department, Rey Juan Carlos University of Madrid, Spain. Currently, he heads the Software Architecture & Internet Technologies (SAIT) research group and leads the IEEE Computer Science Spanish chapter. He is a co-author of more than 80 conference, journals, and book chapters, and also a co-author of the Springer book (2013) *Systems and Software Variability Management. Concepts, Tools, and Experiences*. His research interests focus on software architecture and architecture knowledge, software product line engineering, SOA & cloud computing, and technical debt among others. He also serves as a regular reviewer in conferences, top magazines, and journals. He co-organized several international workshops and is a general chair of the XIV European Conference on Software Maintenance and Reengineering (2010). He is a senior member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.