Towards an Ontology-Based Approach for Deriving Product Architectures

Hector A. Duran-Limon  
University of Guadalajara  
CUCEA  
Jalisco, Mexico  
duran@cucea.udg.mx

Francisco E. Castillo-Barrera  
University of Guadalajara  
CUCEA  
Jalisco, Mexico  
ecastillo@uaslp.mx

Roberto E. Lopez-Herrejon  
Institute for Systems Engineering and Automation  
Johannes Kepler University  
Linz, Austria  
roberto.lopez@jku.at

ABSTRACT
Software product line (SPL) engineering has proven to improve software quality and shorten costs and development time. An important aspect in the product line development process involves variability, which is the ability of a system for being customised, changed, or extended. Approaches are required for modelling and resolving variability as well as for verifying the selections. In this paper, we outline our ongoing research towards an approach that automates the derivation of product architectures from an SPL architecture. The proposed approach relies on ontology-based reasoning and model-driven techniques, the former supports the validation of the generated architectures and the generation of the transformation rules while the latter realises the actual target product architectures. We sketch our approach with a voice over IP case example.

Categories and Subject Descriptors
I.2.4 [Artificial Intelligence]: Knowledge Representation Formalisms and Methods—representation languages; D.2.11 [Software Engineering]: Software Architectures—languages

1. INTRODUCTION
There has been increasing interest in modelling variability at the software architecture level [1, 2, 3, 4, 5]. Software architecture [6] 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) [7] 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.

Most approaches to product derivation focus on deriving product implementation. Nevertheless, there have been some efforts addressing product architecture derivation. These approaches [8, 2, 3] are mainly based on propositional logic. Some other approaches such as [9] rely on Model-Driven Development (MDD) techniques [10]. In the former, each variant element has to be accompanied with Boolean formulas, whereas in the latter the domain design is specified in terms of transformation rules. Specifying product derivations in such approaches can be complex and error prone.

The work presented in this paper shows our preliminary results on an approach whose goal is to automate and simplify the generation of a product architecture from a Software Product Line (SPL) architecture. We use ontology formalism to reason about product architecture derivation based on feature selection. Although product architecture derivation has already been analysed by using propositional logic [8, 2, 3], we argue that ontologies bring in more expressive power resulting in shorter and less complex descriptions. Ontologies [11] are an approach to represent knowledge in a hierarchical and structured way. In our case, ontologies capture knowledge of features, constraints, and the semantic relationships between features and architectural elements. An example of such semantic relationships involve relating features to architectural variation points. Ontology-based reasoning engines permit performing automated analysis over ontologies. We use such engines to determine the architectural elements of a feature selection and drive the generation of model transformation rules that perform the actual specific product architecture derivation from the SPL architecture.

Development of tool support is underway. Currently, we have built an interface to query an ontology-based reasoning engine. An SPL architecture editor has been built in the Eclipse’s GMF plugin [12]. We have also developed a case example, whereby the semantic relationships between the feature tree and architectural elements are captured in an ontology in the Protégé editor [13].

2. ARCHITECTURE CONFIGURATION FRAMEWORK
The overall approach of our product architecture derivation engine is depicted in Figure 1. The feature model, the
Figure 1: Overall Framework Approach

constraints, and the semantic relationships are transformed into an ontology expressed in OWL [13]. The reasoning engine receives as input the ontology and a query based on the selected features. The reasoning engine infers the architectural elements that realise the selected features in terms of pair values involving an architectural variation point and its selected architectural variant(s). The rule generator produces the transformation rules in charge of transforming a variation point into zero or more variants. We distinguish two levels of languages: the product language which corresponds to an ADL description that specifies concrete products, and the SPL language which is an ADL that supports variability modelling. In our framework, we consider the product language is a subset of the SPL language.

A model-to-model transformation engine is then in charge of transforming the architecture of an SPL specified in an SPL language into a specific product architecture described in a product language. The metamodels of the SPL language and product language are defined in the Ecore language [14].

Our proposed modelling process involves two stages: preparation and domain modelling. The first stage is carried out anytime a new SPL language is used. In this stage, the metamodels of the SPL language and product language are specified in Ecore. This stage is necessary to carry out the transformation of a model in the SPL language to a model in the product language. The domain modelling stage consists of the following steps:

1. Define the feature model and feature constraints in a feature modelling tool.
2. Define the SPL architecture in the SPL language using a visual editor.
3. Define the architectural semantic relationships and the architectural constraints. For instance, the architectural variation points and variants are defined. Such relationships and constraints can be represented in an orthogonal variability model (OVM) [15]. Tool support then transforms the feature tree and the OVM diagram into OWL represented in Turtle 1, which is a textual concrete syntax of OWL. Alternatively, instead of using an OVM diagram, relationships and constraints can be directly represented in Turtle.

4. Given a selection of features, the tool transforms the SPL architecture design into a product architecture design.

Below we provide some details of the modules included in the product architecture derivation engine. It is important to mention that we are using Pellet2 as the ontology-based reasoning engine and an ATL transformation engine [16] as the model-to-model transformation engine.

2.1 Ontology Generation Engine

As said earlier, ontologies are used to model features, constraints, and architectural semantic relationships. The ontology generation engine is underway. Ontology instances, represented in OWL, will be generated by using model-to-text transformation techniques. We will use the tool defined in [17] as the feature tree editor. This tool was developed with MDD techniques and represents a feature tree in an Ecore model. We plan to use Accelero3 to transform the Ecore model into Turtle.

In an ontology, features and architectural elements are represented as classes. OWL is based on description logic (DL) [18], which is a family of logic-based knowledge representation formalisms. The domain is described in terms of individuals, classes, and properties. The general structure of all ontology instances include the following elements, as depicted in Figure 2:

1. Features: defines parent relations of the feature tree.
2. Commonality: defines the architectural elements that are common.
3. Variation point: defines the architectural elements that correspond to variation points.
4. Variant: defines the architectural elements that correspond to a variant.

The generated ontology is employed by the reasoning engine to infer which variant architectural elements must be instantiated at a variation point.

2.2 Query Interface

The query interface module is in charge of producing ontology queries based on feature selections. This module was developed in Java and uses the OWL API library4 to perform queries on the ontology-based reasoning engine. The query interface receives as input the root feature and the

1http://www.w3.org/TeamSubmission/turtle/
2http://pellet.owlql.com
3http://www.accelero.org
4http://sourceforge.net/projects/owlapi/
selected features that are leaves in the feature tree. The output generated by the query interface involve four types of queries defined in the Manchester OWL syntax\(^5\), which is a user-friendly syntax for OWL descriptions.

The commonalities (i.e. the architectural elements that do not vary) are obtained by the following query: \(\text{isCommonalityOf some feature}_j\). This query retrieves all the individuals that have the property of being a commonality in the SPL whose root feature is \(j\). It is possible to obtain the architectural variation point associated with a particular feature with the following query, which retrieves the individual that has the property of being a variation point of feature \(i\): \(\text{isVariationPointOf some feature}_i\). The following query determines whether a specific feature has any dependencies with any other features, that is, the query retrieves all the individuals (i.e. features) that have the property of being dependent of feature \(k\): \(\text{isDependentOf some feature}_k\). Finally, the query \(\text{isVariantOf some feature}_m\) retrieves all the individuals (which in this case are architectural variants) that have the property of being a variant of feature \(m\). An example of an example of an ontology query is presented below.

2.3 Rule Generator Engine
The rule generator engine is in charge of producing ATL rules [16]. The development of this module is underway. The rule generator will receive as input the architectural elements that are commonalities and the architectural elements that are variabilities, which are generated by the ontology-based reasoning engine as a result of the queries performed by the query interface. Such variabilities involves a set of tuples, in which the first element of the tuple is a variation point and the second element is a set of selected variants that will instantiate the associated variation point. A variation point defines the parts of the system that may vary whereas the variants represents the options that can realise a variation point. The rule generator is also in charge of invoking 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.

3. CASE EXAMPLE
The case example consists of an SPL for voice over IP (VoIP) systems, as shown in Figure 3. For communicating voice efficiently different compression (and decompression) strategies for data streams are required, e.g. GSM, LPC10, and MELP [19], etc. The presentation style of the user interface can be either text-based or graphical. The user interface can support different languages, namely English, Spanish, or French. Also, it can be selected one form of communication: one-to-one or many-to-many. Finally, a product configuration can be supported by one of the following operating systems: Windows, Mac OS, or Linux. Later when the domain design is defined in an SPL language, the architectural variation point and variants are defined in an OVM diagram. Architectural variation points are mapped to features. For example, the variation point \(\text{vp}_\text{codec}\) is mapped to the feature MELP.

3.1 PL-Xelha: An SPL language
\(^5\)http://www.w3.org/TR/owl2-manchester-syntax/

Figure 3: Feature Model Tree

Xelha [20] 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. There are two kinds of interfaces: provided and required. The former offer services whereas the latter requires them. Therefore, required interfaces can only be bound to provided interfaces. Components are represented as UML components whereas connectors are depicted as UML classes stereotyped as connectors and customised as rectangles. Solid lines denote the commonalities of an architecture while dotted lines denote the variabilities. A variation point can be represented either by a component or a connector. An optional variation point and a multi-optional variation point are represented together with their interfaces by dotted lines and stereotyped as \(<<\text{op}_\text{vp}>>\) and \(<<\text{m-op}_\text{vp}>>\) respectively. In contrast, alternative variation points and multi-alternative variation points have their interfaces denoted by solid lines and are stereotyped as \(<<\text{alt}_\text{vp}>>\) and \(<<\text{m-alt}_\text{vp}>>\), respectively.

3.2 The Ontology of the VoIP SPL
Once defined the feature model and the OVM diagram, shown in Figure 3, an ontology would be automatically generated. Part of the customised ontology that we developed is depicted in Figure 4. There are two variant classes, namely \(\text{LinuxVariant}\) and \(\text{MelpVariant}\). The former contains the codec, presentation style, and language variants for Linux whereas the latter contains the MELP variants for the three operating systems. There are also a number of properties (i.e. relationships) defined. For instance, \(\text{MelpVariant}\) is variant of \(\text{Melp}\) whereas \(\text{LinuxVariant}\) is a variant of \(\text{Linux}\). Also, \(\text{MelpVariationPoint}\) is a variation point of \(\text{Melp}\).

A number of queries can be performed on the ontology to obtain both the variation points and associated variants that realise the selected features. As an example consider a number of feature selections are carried out including \(\text{Melp}\) as the compression scheme for the codec, and \(\text{Linux}\) as the operating system platform. A list of features in which \(\text{Melp}\) has a dependency is obtained by performing the following query for each selected feature that is a leaf: \(\text{isDependentOf} \)
some feature. The variant(s) that realises Melp given the selected features is(are) obtained by carrying out an intersection with the classes in this list. In the case of Melp, only a dependency with the feature Linux was found, hence, the query to obtain the variants of Melp is formulated as follows:

isVariantOf some Melp and isVariantOf some Linux

The result of this query is c_melp_linux. Hence, in the case of the variation point associated with the feature Melp the ontology-based reasoning engine produces the following output: \{vp_codec, \{c_melp_linux\}\}. The element c_melp_linux is the variant that instantiates the variation point vp_codec. In contrast, a description similar to the following is required when using propositional logic to determine whether the architectural element c_melp_linux is selected or not:

c_melp_linux requires (Melp and Linux)
and (c_graphical_linux or c_textual_linux)
and (c_onezone_linux or c_many2many_linux)
and (c_english_linux or c_spanish_linux or c_french_linux)

In this example, the expressive superiority of DL derives from the fact that the propositional formula performs Boolean operations on single elements whereas the DL expression carries out operations on sets of elements. Furthermore, each variant element has to be accompanied with Boolean formulas when using the propositional logic approach. In contrast, in our approach a DL expression is only needed for each architectural variation point. Therefore, more descriptions are needed when using propositional logic since the number of variants is larger than the number of variation points. Finally, the DL expressions to query the ontology are automatically generated by the query interface whereas the propositional formulas need to be elaborated by the user.

3.3 Deriving the Product Design

As an example of SPL architecture modelling, the connections of the vp_codec variation point are represented in PL-Xelha, as shown in Figure 5. It should be noted that the variation point involves a component. Moreover, the components c_src_stub and c_sink_stub are connected to the variation point. The output generated by the ontology-based reasoning engine is received as input by the rule generator engine in charge of generating the model-to-model transformation rules. The ATL language [16] will be used for specifying the transformation rules. The transformation engine is in charge of executing the generated rules whereby the product design model is obtained.

Figure 5 shows the product architecture that would be generated in our case example. The SPL model is transformed to a product model whereby the variants are removed. The component c_melp_linux takes the place of the variation point component.

4. RELATED WORK

Most approaches to product derivation focus on deriving product implementation. However, a few works have addressed the issue of deriving product architecture. There have been some efforts that allow for automatically generating a component configuration based on the features selected by the user [3], [2], [8]. Nevertheless, applying the above approaches to large-scale systems involving hundreds of features and variants is a complex and error-prone task since each variant element has to be accompanied with special annotations with Boolean formulas. In contrast, in our approach, the knowledge required to determine whether a variant should be included or not is not defined in the architecture model but in the ontology, thus, achieving a separation of concerns. Furthermore, we argue that our approach representing the knowledge for deciding whether to include or not a variant is simpler. The above approaches are based on propositional logic whereas our approach is based on DL which has more expressive power [18]. Some other approaches such as [9] employ model-driven techniques to transform a feature model to specific product architectures. However, the domain design is specified in terms of ATL transformation rules [16], therefore the transformation pro-
cesses is not completely automated. Such an approach is complex and makes the SPL architecture design process difficult. The approach presented in [21] also supports model-driven transformation of model features. However, their approach focuses on product implementation derivation without deriving the product architecture.

5. CONCLUSIONS

We have presented the outline of our approach to the derivation of product architectures. Different from other works which are based on the use of propositional logic, ours relies on DL. Consequently, our framework is able to simplify the descriptions of architectural constraints. At the core of our approach is an ontology-based engine. This engine enables reasoning about feature selections and the validation of architectural constraints, while at the same time guides model-driven techniques to generate the final product architecture for the selected features. So far we have built an interface to query an ontology-based reasoning engine. A visual editor of SPL architectures was developed in the Eclipse GMF plugin. We have also designed the generic ontology structure that all ontology instances must obey. Our ongoing work involves tool support for transforming a feature tree and an OVM diagram into OWL, and a prototype of the rule generator engine. Most salient, an evaluation with large-scale and complex architectures is necessary.

Acknowledgments. This research was partially funded by Marie Curie Actions - Intra-European Fellowship (IEF) project number 254065.

6. REFERENCES


