Management of Variability in Modular Ontology Development

Melanie Langermeier[1], Peter Rosina[1], Heiner Oberkampf[1,2],
Thomas Driessen[1], and Bernhard Bauer[1]

[1] Software Methodologies for Distributed Systems, University Augsburg, Germany
[2] Siemens AG, Corporate Technology, Munich, Germany

Abstract. The field of variability management deals with the formalization of mandatory, alternative and optional domain concepts in product line engineering. Ontologies in turn, describe domain knowledge in form of predicates, subjects and constraints in various forms. Based on existing ontology mapping approaches, we developed a method to organize a set of modular ontologies using the concepts of variability management (MOVO). This ontology driven variability model can be stepwise adapted to the needs of a business driven one, resulting in a variability model that fits the needs of business and makes modular ontologies reusable in a simple manner. In order to avoid a technological break and to benefit from the opportunities that ontologies offer, the resulting variability model is expressed in an ontology itself. The approach is evaluated by one case study with enterprise architecture ontologies.

Keywords: Modular Ontology Management, Variability Management, Feature Models, Ontology Mapping.

1 Introduction

Knowledge management (KM) is a central aspect in organizations. KM tools mainly rely on knowledge models, specifying how knowledge is represented. In general, there is not one single knowledge model which could be used within all applications or tools. In contrast, the knowledge models used in KM tools are largely dependent on the application and/or the customer's/department's needs. Within one organization, different departments might need different models to describe their knowledge, even though parts of their models describe similar aspects. Since the creation of knowledge models is costly and error prone, it is desirable to reuse existing knowledge models which have proven to be useful and only customize them to specific needs. Modular developed knowledge models allow to reuse parts and ease the customization. Similar to software products, knowledge models have certain logical and functional dependencies. What we need, is a mechanism to make these dependencies between modules of knowledge models transparent in order to enable a flexible combination of them.

In classical software product lines, the variability management has proven to be useful. Thereby, commonalities and differences between domain concepts are made explicit and allow an effective management of the variability in the product development process. This systematical approach enables a consequent reuse of existing concepts and

makes possible combinations and dependencies transparent [7]. This general-purpose and reusable methodology is not only applicable to software products.

Ontologies, and in particular the Web Ontology Language OWL 2[1], have become very popular for the representation of knowledge. Ontologies offer a flexible and powerful way to represent a shared understanding of a conceptualization of some domain [15]. Efficient reuse of (parts of) ontologies is one of the main goals behind modular ontology development [24]. OWL offers some mechanisms, such as the `owl:import` relation, to combine and integrate ontologies. The use of logical axioms contained in different ontologies, however, restricts the possible combinations. A formalism representing these restrictions is needed.

The management of possible combinations of modular knowledge models is similar to the management of variabilities in software products. In this paper, we describe the use of variability management for the management of modular ontologies (MOVO), i.e., to describe the logical and functional dependencies between ontologies. Each possible variant is described through a set of features, that are linked to ontology modules. In our scenario, the knowledge engineer (KE) iteratively selects ontology modules from the ontology repository and creates the variability model (VM) using these ontology modules. In each iterative step, validations are run on the ontology to check the consistency of the VM. The resulting VMs are realized with feature models, formalized in OWL, and stored in a VM Repository. They are instantiated to create specific customized application ontologies. Formalizing feature models with OWL avoids a technology break and enables the use of reasoning capabilities to support the KE by detecting and dissolving inconsistencies in individual and aligned ontologies. We evaluate our approach with an enterprise architecture (EA) case study.

The remainder of the paper is organized as follows: in section 2 we describe the state of the art in modular ontology development and variability management, followed by an overview of related work in section 3. Based on this, we describe our proposed method in section 4 and their technical realization in section 5. The evaluation is done with the aid of our use case in section 6, before we conclude our work in section 7.

2 State of the Art

2.1 Modular Ontology Development

Large ontologies have certain disadvantages regarding reuse and performance. Modular ontology development tries to overcome these obstacles. The general idea is to keep ontologies small in creating ontology modules focusing on one particular aspect to enhance (partial) reuse and performance (e.g. more efficient reasoning), ease maintenance (smaller ontologies are easier to comprehend) and collaborative development as well as harmonization and interoperability (using common upper ontologies, it is easier to identify mappings). Details can be found in [24] and [25]. These modules themselves are again ontologies [12]. Application ontologies, which have to cover different topics, are created using several small ontologies (modules). A number of different promising approaches have already been investigated and evaluated [25]. In accord with [21], modularized ontologies cover two separate topics: (1) module extraction (i.e., modularization

[1] `http://www.w3.org/TR/owl2-overview/`

of existing large ontologies into smaller logically consistent modules) and (2) modular development (i.e., the creation of modular (non-redundant, orthogonal) ontologies). Our work targets the management of ontology modules and their composition into one application ontology. With OWL 2, we have a standardized vocabulary for the description of ontology meta data such as the ontology IRI, `owl:versionInfo`, or `owl:versionIRI` and relations between ontologies such as `owl:imports`, `owl:backwardCompatibleWith`, `owl:incompatibleWith` or `owl:priorVersion`. All relations between ontologies, except `owl:imports`, are annotation properties and have only a documentation purpose or describe functional dependencies.

Similarly, the Vocabulary of a Friend ontology[2] (VOAF) defines properties to express relations between RDFS vocabularies or OWL ontologies: for instance, `voaf:reliesOn`, `voaf:extends`, `voaf:specializes` or `voaf:generalizes` can be used to *indicate* how some ontology is related to others. Besides relations between ontologies, OWL also offers relations between concepts of different ontologies, e.g., stating equivalence between individuals and classes respectively. These relations are regarded by standard reasoners. With `owl:imports`, other ontologies can be included into an ontology through reference onto the other ontology's IRI. Critic concerning this approach has come up, because it is not possible to only import parts of another ontology, but only all the axioms of the other ontology. Therefore, if just a subset of another ontology is needed, a modularization can be helpful. Simply referencing single entities of another ontology, without using the `owl:imports` construct, does not transfer its semantics and context. Working with modules leads to ontology mappings to align different modules. According to [8] there are three types of mappings: *i.)* mapping between one integrated global ontology and various local ontologies; *ii.)* mapping between different local ontologies; and *iii.)* ontology merging and alignment. Since our work focuses on the management and composition of ontology modules, we mainly deal with the second and third type of mappings.

2.2 Variability Management

The discipline of variability management deals with the consequent and explicit documentation of the variability of software artifacts in product line engineering [7]. Variability is the "ability of a system or artifact to be extended, changed, customized, or configured for use in a specific context" [23]. Through a consequent and explicit representation of variabilities using variability modeling techniques, the software engineers are able to manage those and thus complexity in the development process can be reduced [23]. An overview of variability techniques can be found in [7] and [23]. One of the major benefits is the systematic reuse of existing artifacts [6,18,7]. The development of a product in product families is done in two steps: first, in the domain engineering, the commonalities and differences of the products are determined and a set of reusable artifacts like a product family architecture and a set of components is created. Second, during application engineering, the final products are build through configuration of the reusable artifacts [23,6].

Features are a widely used concept for the identification and documentation of variabilities. In the context of software product lines, they are defined as logical units of

[2] `http://purl.org/vocommons/voaf`

behavior that are visible to the end-user [13,3]. These features are encapsulated within components at the architectural level and thus enable an easy inclusion or exclusion of single components [18].

Kang et al. introduced in 1990 the Feature-Oriented Domain Analysis (FODA) with the first Feature Models. They replaced the formerly used sequence diagrams. The intent of the author was "to capture in a model the end-user's (and customer's) understanding of the general capabilities of applications in a domain" [17]. Figure 1 depicts the graphical notation of Kang's feature models. The different features of a domain are structured in Parent-Child-Relationships, which result in a feature tree. Depending on the connection between parent and child, the semantics between both is defined as follows:

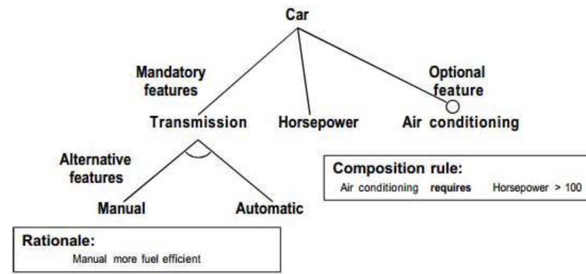

Fig. 1. Graphical notation of the FODA Feature model [17]

Optional Feature:	defined by a line with a circle at its end; can, but has not to be, chosen, if the parent feature is selected.	
Mandatory Feature:	defined by a line without an additional decoration; has to be selected, if the parent feature is selected.	
Alternate Feature:	defined by two or more lines, that are connected via an arc; exactly one of those has be chosen if the parent feature is selected.	
Composition Rule:	dependencies between features of different sub-trees, that can not be expressed in the hierarchical way of the feature tree.	
	Requires:	a feature has to be selected, based on the selection of another feature.
	Mutually exclusive with:	a feature must not be selected if another feature is already selected.

This kind of feature model is restricted to the analysis phase of a software project. Kang et al. extended his approach to the design phase of a project (Feature-oriented reuse method, FORM) [16]. In order to be able to reference possible implementations of a feature in code, Kang et al. introduced the concept of layers, explicit generalizations and an implemented-by reference. Czarnecki et al. extended the FODA Feature model with concepts for the assignment of cardinalities to features and feature groups, the assignment of data types to features and the definition of references from features to the root of another feature tree [10].

3 Related Work

The state of the art regarding modular ontologies, as well as mappings, are described in section 2.1. There, we also described existing vocabularies for relations between ontologies like, e.g., those provided by OWL or VOAF. In this section, we describe related work regarding the use of ontologies to represent feature models and to use established

reasoning mechanisms to validate them. The expressiveness of feature models (FM) in comparison to ontologies is analyzed, for instance, in [11]. They identified, that basic feature models are less expressive than OWL ontologies. However, there exist several extensions of basic feature models enhancing the expressiveness, e.g., the addition of attributes, the cloning of entities or feature value constraints. [26] describe, which requirements should be fulfilled by a Semantic Web technology-based feature model: *automated inconsistency detection*, *reasoning efficiency*, *scalability*, *expressivity* and *debugging aids*. They state, that "OWL can be adopted to reason and check feature models effectively". OWL DL syntax is used to represent feature models, where feature nodes are represented as OWL classes. They demonstrate that all of the standard feature model relations (mandatory, optional, alternative, or) as well as simple constraints (excludes, requires) can be represented. Similarly, [27] use OWL DL to represent feature models, even though the modeling is significantly different to [26]. In [27], classes are used to represent features, compositions, feature attributes and feature relations. OWL properties are used to represent feature to feature constraints, attribute value constraints and compositional properties. The consistency is checked using an OWL DL reasoner and SWRL rules, e.g., the mutual exclusiveness of certain properties. In summary, the model presented by [27] is even more expressive than the one presented by [26]. Another approach of using ontologies for modeling variability in a product/service family domain is presented in [19]. In this approach not only the variability itself is captured in an ontology, but also the reasons that led to the respective variability point.

[20] use FMs and ontologies to support the selection of features in multi-cloud configurations. Their method proposes to create the FM first, then map a cloud (the domain) ontology's concepts to the FM's features until every connection is established. These procedures are performed manually by domain experts. Afterwards, they are validating their model. In contrast to our approach, they are using EMF meta models, resp. XMI models, that represent their FMs, as well as their ontologies and mapping models. This way, ontological (OWL) reasoning cannot be performed, but they propose using a SAT solver, for instance Sat4j [2], for checking the FM's configuration validity.

4 Method

In the following, we propose our method for the management of variability in modular ontology development (MOVO). This method aims to address the issues of modular ontology development described in section 3. Thereby, our method focuses on dealing with the complexity of mandatory and optional dependencies between the single modules as well as mandatory exclusions between them. Figure 2 shows the main concepts of MOVO as well as the two phases of the method. In the first step, the KE has to select the modular ontologies, which will be stored in the ontology

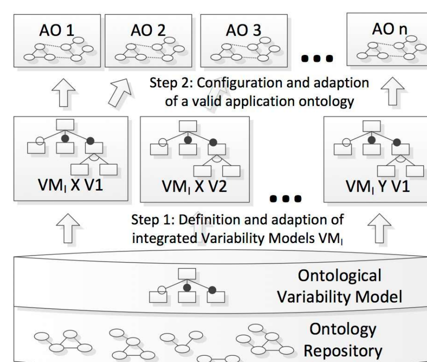

Fig. 2. Overview of the concepts in MOVO

repository. Based on those ontologies an ontological variability model VM_O is defined. VM_O formalizes the dependencies between the modular ontologies that are annotated in the ontologies. It defines allowed and not allowed variants of the application ontology. The variants are defined using features which could/should be (not) included. Each feature can, but has not to, be linked to one or more modular ontologies. Based on VM_O, the KE creates VM_I through selection of features and relationships and addition of stronger constraints according to specific domain requirements. This model formalizes the dependencies according to the requirements from the domain while considering the ontological restrictions. In other words, it formalizes which variants make sense and which not in combination with what is allowed and what is not allowed.

VM_O formalizes the dependencies annotated in the ontologies whereas VM_I customizes these constraints according to specific domain requirements. We differentiate between these models to be able to differentiate between ontological and domain specific requirements and therefore enable the creation of several VM_I for different domains upon one set of ontological modules. The method ensures that the created VM_I is consistent according to the `owl:import` and `owl:incompatibleWith` assertions, which can be made in the single ontological modules. After the creation of a consistent VM_I by the KE, the domain expert can easily create consistent configurations for his application ontology. Figure 3 illustrates the relationships between the concepts used in MOVO.

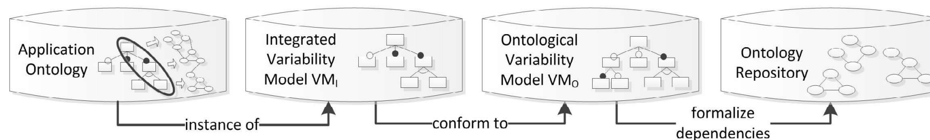

Fig. 3. Concepts and their relationships required for the definition of the VM

In the following, the definition of the ontological VM, along with the creation of the ontology repository, the definition of the integrated VM as well as the configuration of an application ontology, are described in more detail. The technical realization of those steps is described in section 5.

4.1 Define Ontological Variability Model VM_O

Before the ontological variability model VM_O can be determined, the KE has to fill the ontology repository. There are several sources for modular ontologies: reuse of existing ontologies, modularization of existing bigger ontologies or creation of new ontologies. Creating mappings between different vocabularies can either be done manually or with the assistance of automated methods. These methods for matching heterogeneous resource models with semantic technologies are introduced and explained in [22].

In the next step, the meta data and assertions of the modular ontologies in the repository are analyzed to determine the dependencies between them. For this work we decided to focus on the assertions that can be realized using OWL 2. These are the `owl:import` and the version informations. Whereas from the later one only the `owl:incompatibleWith` has effects for the definition of consistent variants. At the moment, OWL 2 does not offer an annotation property that expresses an inconsistency

between two different ontologies. Therefore, we introduce an `movo:inconsistent` relationship, to be able to assert such an information. To create VM_O for each modular ontology, one feature will be created with a link to the corresponding ontology. The dependencies between the ontologies are then formalized in the ontological VM. This model is consistent in sense of allowed combinations of the modular ontologies, but it must not necessarily fulfill certain requirements of the domain. This newly created VM_O acts as the bootstrapping VM for the following creation of VM_I.

4.2 Define Integrated Variability Model VM_I

The integrated Variability Model VM_I can extend and restrict VM_O to be conform to specific requirements of the application domain. Thereby, new features or relations can be added and existing relations between features can be strengthened. For the creation of VM_I, the KE has to select a root feature(existing from VM_O or new one), and then repeats the following loop until all desired features are considered.

i.) Select a parent feature from VM_I or a new one
ii.) Select a child feature from VM_O or VM_I or a new one
iii.) Determination of valid relations that can be used to connect those features
iv.) Select the new features' type of relation
v.) Automatic addition of the features with their relations to VM_I
vi.) Optional: add further cross-tree constraints

Cross-tree constraints can be necessary, for example, when defining that a specific mapping ontology *OntA2OntB* should be always used for two ontologies *OntA* and *OntB*. In this case, the constraint $OntA \wedge OntB \rightarrow OntA2OntB$ is necessary to ensure that the mapping ontology *OntA2OntB* is selected when *OntA* and *OntB* are selected. Finally the KE has an integrated VM which represents all allowed and useful variants of the application ontology.

4.3 Configuration of a Specific Knowledge Model

Preliminary for this step is the defined VM_I. The domain expert is then able to create a specific configuration which serves as an application ontology. Therefore, he selects those features from a list of selectable features he wants to have included in his configuration. After each feature he selects, a consistency check will take place. First, it will be checked, if there is any required feature that is not yet included in the final configuration. If so, then this feature will be included. Second, after the addition of a feature, all features that are excluded by this feature will be deleted from the list of selectable features. At the beginning, the list of selectable features SF includes all features that are in VM_I: $SF := \{f \mid f \in VM_I\}$. The list of selected features in the configuration C is empty at the beginning. If a feature f from SF should be inserted into C the insert function is defined as followed:

$$\begin{aligned} insert(f) := {} & addToConfiguration(f) \wedge removeExludedFeatures(f) \wedge \\ & (\forall reqF.((reqF \in SF \wedge f \rightarrow reqF) \rightarrow insert(reqF)) \\ \text{With} \quad & addToConfiguration(f) : C := C \cup \{f\} \\ & removeExludedFeatures(f) : SF := SF \setminus \{exclF \mid exclF \in SF \wedge f \rightarrow \neg exclF\} \end{aligned}$$

SF ensures that only features can be selected that fit to the current state of the configuration. The last point ensures that no feature will be dismissed that is required by a selected feature in the configuration. After the domain experts has defined his configuration of features, the corresponding ontologies to the features have to be selected and composed to the resulting application ontology.

5 Technical Realization

Along with our method described in the previous chapter, we describe the technical realization of the two main steps: the creation of the ontological and integrated variability model (VM_O and VM_I) and the instantiation of VM_I. The technical realization is exemplary demonstrated using the FODA feature model from [17] described in section 3. In this context, we are using Protégé[3] to create our ontologies and a Fuseki Server[4] as a triple store for our prototype implementation.

5.1 Variability Model Ontology

The VM_O is specified using OWL 2 semantics. The central class of the VM_O is `movo:Feature`. An instance of `movo:Feature` represents a node of the VM and might be related to some ontology of the ontology repository using the object property `movo:isRealizedIn`. As described in section 3, OWL and especially OWL 2 specify several annotation properties for meta information of ontologies and relations between ontologies. Some of them are shown in figure 4. These annotation properties are not interpreted by reasoners, thus the idea is to translate these annotation properties (which describe the coherence between different ontologies) to object properties in VM_O. For instance, we defined the object properties `movo:excludes` (for `owl:incompatibleWith`) and `movo:requires` (for `owl:imports`). The property `movo:excludes` is a symmetric property and is mutual exclusive with `movo:requires` (using `owl:propertyDisjointWith`).

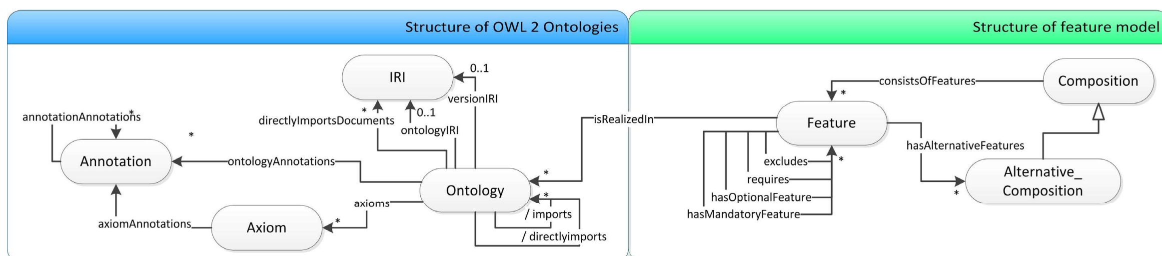

Fig. 4. The Variability Model Ontology combined with OWL 2 Ontology structure [4]

This is similar to ideas presented in [27] where "*Incompatible* and *Excludes* are defined as symmetric properties. Some are mutual exclusive: (*Requires*, *Excludes*), (*Requires*, *Incompatible*), (*Uses*, *Excludes*), (*Extends*, *Incompatible*)". Other OWL annotation properties like `owl:backwardCompatibleWith`, `owl:priorVersion`,

[3] `http://protege.stanford.edu/`; 11/09/2013
[4] `http://jena.apache.org/documentation/serving_data/`; 11/09/2013

owl:deprecated, etc. can be used or realized in VM_O as well and will be considered. In addition to the representation of OWL properties, VM_O has to capture feature model semantics presented in section 2 so that the KE can express further dependencies. For instance, we define properties for optional and mandatory properties, i.e., the relations movo:hasOptionalFeature and movo:hasMandatoryFeature. Again, these properties are mutually exclusive. In our prototype implementation we are using the movo:hasMandatoryFeature and movo:requires relations as logically equivalent properties, because the difference is only important for the graphically distinguished visualization as a feature model tree for the user.

Furthermore, we have a class movo:Composition with the subclass movo:Alternative_Composition to represent alternative compositions (AC). When the AC is created, the source feature is related to the movo:Alternative_Composition via the object property movo:hasAlternativeFeatures. Other compositions, for instance an *OR* composition can be added. Furthermore, enhancements can be made in order to consider the extensions of the FORM feature model, e.g., the cardinalities. The work of [27] demonstrates that OWL DL in combination with some rule language like, e.g., SWRL can be used to represent even more sophisticated feature model constraints.

5.2 Creation of VM_O with Mapping Semantic

Our prototype implementation is realized using a Apache Jena Fuseki triple store with two data sets, one for the ontology repository (ontrepo) and one for the Variability Model Ontology (vmo). We separate ontologies in our repository using named graphs and use the following procedure to create the ontological variability model VM_O: First, for all ontologies in the repository, instances of movo:Feature are created in the dataset vmo. Second, the dependencies between ontologies, such as *import* relations, are transferred to relations between features (see listing 1.1).

```
INSERT {
    ?feature a movo:Feature ;
    movo:isRealizedIn ?ont ;
    movo:requires ?req .
}
WHERE {
    SERVICE <http://localhost:3030/ontrepo/query> {
    SELECT ?feature ?ont ?req ?excl WHERE {
        ?x owl:ontologyIRI ?ont .
        OPTIONAL { ?ont owl:imports ?r .
            BIND (URI(CONCAT("http://www.ds-lab.org/↩
                ontologies/2013/7/variabilityOntology#", ↩
                strafter(str(?r), "http://www.ds-lab.org/↩
                movo/ea/"))) AS ?req) }
        BIND (URI(CONCAT("http://www.ds-lab.org/ontologies↩
            /2013/7/variabilityOntology#", strafter(str(?↩
            ont), "http://www.ds-lab.org/movo/ea/"))) AS ?↩
            feature)
}}}
```

Listing 1.1. Extract of SPARQL statement example for the creation of the ontological Variability Model VM_O

The feature is related to its source ontology by the `movo:isRealizedIn` property. Furthermore, the dependencies between different ontologies are extracted and interpreted, e.g., `owl:imports` to `movo:requires`, using the same SPARQL statement. For other dependencies, such as `owl:incompatibleWith` or `movo:inconsistent`, we have similar update queries. More precisely, for each relation between ontologies, a corresponding relation is added for the respective features. Since the *import* and *incompatible* relations only exist occasionally, we use the `OPTIONAL` statement for these object properties. In this context, we are substituting the resources' URI paths from the ontology repository's source ontologies' location with the new feature ontology's URI path. Thus, the dependencies between ontologies are transformed to the variability model ontology.

During the creation of VM_O, the OWL reasoner and additional SPARQL queries can be used to check the consistency of the created feature model [26]. For instance, it is checked that there are no features related with contradictory properties `movo:excludes` and `movo:requires` at the same time. We are also using SPARQL queries to receive all dependent features, i.e., the required features of the selected feature and thus can add them automatically to our VM_O. Following the principles of the Semantic Web Stack, it is generally advised to use the Rule Interchange Format (RIF) for expressing rules. For instance, RIF would be suitable for stating complex composition rules. Since up to now, RIF is still immature and tool support is hardly available, we use SPARQL in our implementation to insert relations between features.

5.3 Creation of VM_I

The creation of VM_I is done according to section 4.2. Thereby, the features created for VM_O can be reused, but it is also possible to add new features as place-holder features, that do not yet have a relation to an ontology of the ontology repository. For each new feature, a new instance of `movo:Feature` is created and stored in a named graph for the respective VM_I. To ensure that VM_I is conform to VM_O, the integrated variability model will be defined iteratively. In each step, only consistent constraints can be added to the model. The following pseudo code 1.2 represents this procedure.

```
select ROOT FEATURE root
insert(root)
LOOP
    select PARENT FEATURE p
    select CHILD FEATURE[S] c = {c1, .., cn}
    if (|c| = 1)
         then ask(required), ask(hasMandatoryFeature), ask↩
             (excludes)
         else ask(alternateComposition)
    select POSSIBLE RELATION relation
    for all (x in (c or p); x not in VMI)
        insert (x)
    insert(relation)
    OPT: if(ask(crossTreeConstraint))
                  then insert(crossTreeConstraint)
END LOOP
```

Listing 1.2. The procedure for creating VM_I

To ensure the consistency, the following SPARQL queries are defined:

insert(feature): adds an existing feature from VM_O to VM_I with all (transitively) required features

ask/insert(required|hasMandatoryFeature|excludes): asks if possible or inserts the respective relationship between a parent and one child

ask/insert(alternateComposition): asks if possible or inserts an alternate composition between a parent and a set of childs

ask/insert(crossTreeConstraint): asks if possible or inserts a specific cross tree constraint, typically in a manner like '$feature_1$ requires $feature_2$' or '$feature_1$ excludes $feature_2$'

For selecting a specific feature, the following constraints must be satisfied:

select ROOT FEATURE: root $\in VM_O$ or root is a new feature
select PARENT FEATURE: parent $\in VM_O \cup VM_I$ or parent is a new feature
select CHILD FEATURES: c = c_1, .., c_2 with $c_i \in \{VM_O \cup VM_I\}$ or c_i is a new feature

During the creation of object properties between the selected features, some constraints have to be fulfilled: for the sake of simplicity, we only allow the creation of relations between exactly two different features (an exception is the alternative composition(see below)). Additionally, there can only be exactly one or zero relations between two different features. These constraints will be checked using the *ask* queries. Only when these queries return true, the KE can fulfill an insert of the relationship. Adding the `movo:hasOptionalFeature` or the `movo:requires` relation is only valid if there is no `movo:excludes` between the source feature and an existing transitive `movo:requires` path to the target feature. Adding a `movo:excludes` relation is only valid if there is no transitive `movo:requires` or `movo:hasOptionalFeature` in VM_O or VM_I.

A precondition for the creation of the alternative composition (AC) is the non-existence of any relationship from the source feature to any of its target features. We also forbid a transitive `movo:requires` relation between any two features that are in the set of the AC. Besides, to keep it simple, another constraint is that we do not allow the creation of nested or overlapping ACs. This constraint could be relaxed in the future. When the AC is created, the source feature is related to the `movo:Alternative_Composition` via the object property `movo:hasAlternativeFeatures`. Simultaneously, we add `movo:excludes` relations between all members of the AC, since it represents an XOR selection. The AC is the only existing relation between more than two features. Once VM_I is finished, it is saved in a fresh data store.

5.4 Instantiation of VM_I

The instantiation of VM_I corresponds to the creation of the user configuration C (compare section 4). We create a new data set and add a property to each feature in VM_I that expresses its status: selectable, selected and not selectable. According to the rules already described earlier, we automatically select all required features by querying the transitive paths and disable the not selectable features in case of a `movo:excludes` resp. AC relation. The validation of our user configuration using OWL is not part of this paper, because there already exist some reliable approaches (see, e.g., [26] or [27]). Once the final configuration has been found, the qualified ontologies, including the mappings between them, are deployed as the compound application ontology .

6 Evaluation

The design and implementation of enterprise architecture (EA) methods and analyses are dependent on the meta model used in the organization. Typical for EA is, that each organization has its own meta model for EA. Typical for EA is also, that this meta model depends on already existing ones in the different organization units. For example, the process modelers have their model about processes, the IT administrator has its model about the infrastructure and the software development unit has its meta model about the application landscape. To increase the acceptance of the enterprise architecture in the organization, it should be built with respect to those existing models.

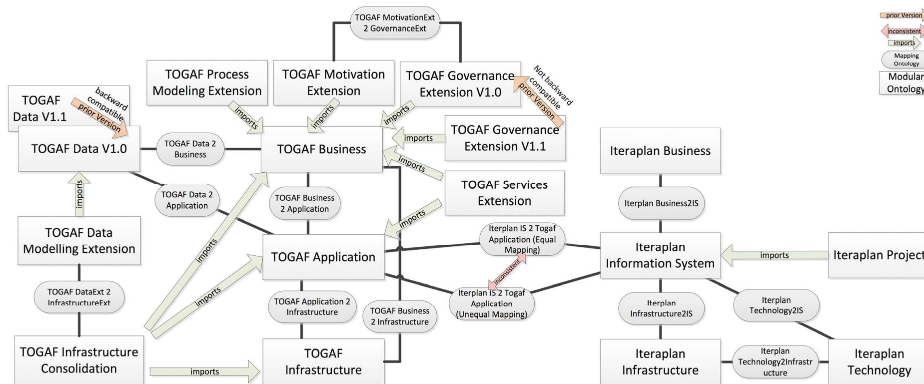

Fig. 5. Modules and their relationships in the EA Use Case

Especially for those providing tool support for EA, this issue is a challenge. On the one hand, the organizations want to rely on existing frameworks and meta models, but, on the other hand, they also want to adapt them to their specific needs. To illustrate this problem, we choose two meta models for enterprise architecture, modularize them, and establish a variant model, which allows a flexible combination of different parts of the meta model. This enables the tool provider, who plays the role of the KE, to establish methods, that support the enterprise architect, independently from the final meta model or with respect to a special selection. The enterprise architect, which will be the domain expert, can easily create his desired configuration which will act as his customized meta model.

For the case study we choose the TOGAF Core Content Metamodel[5], a standard from the Open Group, and the meta model behind the enterprise architecture tool iteraplan[6]. To get modular ontologies, we first divided the two meta models into smaller modules according to the architecture layers the frameworks present. The relationships between these layers are represented through import relationships and mapping ontologies. I.e. there exist two different mappings from Iterplan Information System to the TOGAF Application module, that cannot be used together. All determined modules with the mapping ontologies and imports are shown in figure 5. This set of modules represents the ontology repository. The generated VM_O formalizes the `owl:imports`

[5] `http://pubs.opengroup.org/architecture/togaf9-doc/arch/chap34.html`
[6] `http://www.iteraplan.de`

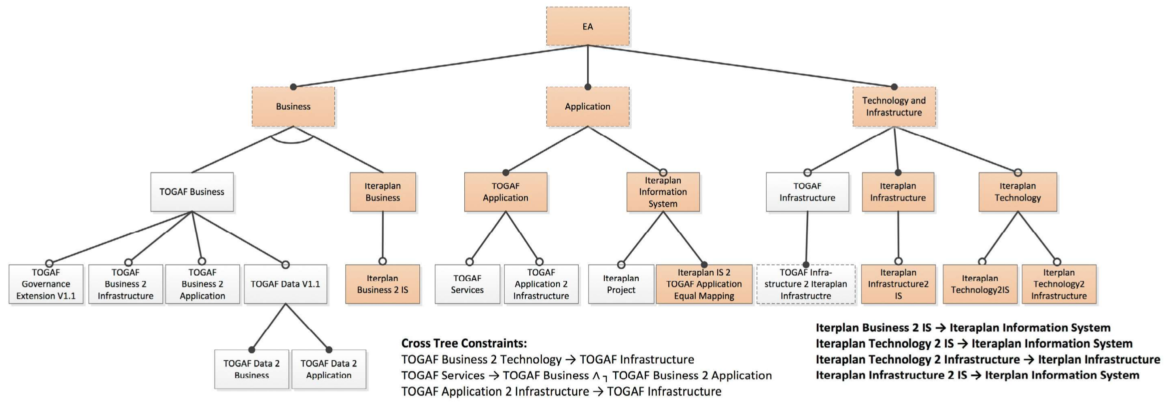

Fig. 6. Feature Model VM_I in the EA Use Case

and `owl:incompatibleWith` relationships. All relationships are correctly transformed into *required* and *exclude* relationships.

For the further evaluation, we define and establish an integrated variability model by selecting the desired features and adding further constraints. The shaded rectangles highlight a possible configuration. The integrated variability model with the configuration is shown in figure 6. VM_I is conform to VM_O and we are able to model all requirements from the domain. All *ask* queries enable us to insert the desired relationships. Additionally, every other required module, that we do not explicitly select, is inserted. To model the alternate choice between the iteraplan business and the TOGAF business module, we create a feature, that is not linked to any ontology. This enables the modeling of a choice between several features. We also introduce such empty features for the other architectural layers, since the resulting feature model is more comprehensive for a domain expert. These empty features are depicted by dashed lines surrounding the rectangles. Furthermore, we introduce one more feature that is not related to any ontology. This ontology has to be added if this feature is selected in a configuration.

Our test set for the evaluation, including the data sets, queries and a documentation, has been published at `http://megastore.uni-augsburg.de/get/HAth0VS7qw/`.

7 Conclusion

In this paper, we proposed a method for the management of modular ontological models. We especially addressed the problem that the dependencies between the single modules can not be specified using the standard OWL vocabulary. We use the concept of variability management in software product line engineering and adapted it to the domain of modular ontology management to be able to formalize possible combinations of the modules. Therefore, we defined a mapping from the OWL concepts `owl:imports` and `owl:incompatibleWith` as well as from `movo:inconsistent` to the concepts `movo:requires`, `movo:excludes`, `movo:hasMandatoryFeature` and `movo:Alternative_Composition` to determine an ontological variability model. Additionally, we provide a method to create an integrated variability model which is, on the one hand, conform to the ontological variability model, which specifies what is allowed and what not. On the other hand, it specifies what makes sense and what not in the resp. business

domain. Based on the integrated variability model, a domain expert can easily create her application ontology through selection of those features she wants to have. The required set of ontologies to create the application ontology can then be retrieved from the variability model using existing feature model solver. To be able to use the reasoning techniques of ontologies we defined the variability model ontology to express the VMs in ontologies.

Our method enables the reuse and flexible combination of knowledge modules in several application ontologies. Thereby, it ensures that the resulting application ontology is conform to the annotations that are made in the ontology modules and also to the requirements that the KE specified. Our goal is to support KEs in assembling a customized ontology set by providing a modeling environment that applies semantic technologies.

Future work has to be done to explicitly provide methods to adapt the variability model when changes in the ontology or requirements for the features have taken place. In our prototype implementation, we are just using two annotations of the given OWL functionality for combining modular ontologies. In the future, we want to cover additional annotation possibilities in OWL, vocabularies like VOAF and also consider more expressive approaches for defining coherences between the ontologies. Alternative approaches, like $\mathcal{E}$-Connections [9], Package Based Description Logics (P-DL) [1], Distributed Description Logics (DLL) [5] or the Interface-based modular ontology Formalism (IBF) [14] are eligible alternatives and extensions for modular ontologies. These approaches offer similar functionalities: they offer bridge rules between multiple ontologies, a specific *point of views* interpretation for modular ontologies or the support for well-defined interfaces between the ontological modules. Therefore, we also want to extend the expressiveness of the method and the variability model ontology, e.g., with an OR composition or cardinalities.

References

1. Bao, J., Caragea, D., Honavar, V.G.: Modular ontologies - A formal investigation of semantics and expressivity. In: Mizoguchi, R., Shi, Z.-Z., Giunchiglia, F. (eds.) ASWC 2006. LNCS, vol. 4185, pp. 616–631. Springer, Heidelberg (2006)
2. Berre, D.L., Parrain, A.: The Sat4j library, release 2.2. JSAT 7(2-3), 6–59 (2010)
3. Beuche, D., Papajewski, H., Schrder-Preikschat, W.: Variability management with feature models. Science of Computer Programming 53(3) (December 2004)
4. Bock, C., Fokoue, A., Haase, P., Hoekstra, R., Horrocks, I., Ruttenberg, A., Sattler, U., Smith, M.: OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax. Tr, W3C (2009)
5. Borgida, A., Serafini, L.: Distributed Description Logics: Assimilating Information from Peer Sources. Journal on Data Semantics 1, 153–184 (2003)
6. Bosch, J., Florijn, G., Greefhorst, D., Kuusela, J., Obbink, J.H., Pohl, K.: Variability issues in software product lines. In: van der Linden, F. (ed.) PFE 2002. LNCS, vol. 2290, pp. 13–21. Springer, Heidelberg (2002)
7. Chen, L., Babar, M.A., Ali, N.: Variability management in software product lines: a systematic review. In: Proceedings of the 13th International Software Product Line Conference, SPLC 2009, pp. 81–90 (2009)
8. Choi, N., Song, I.-Y., Han, H.: A survey on ontology mapping. SIGMOD Rec. 35(3), 34–41 (2006)

9. Cuenca Grau, B., Parsia, B., Sirin, E.: Ontology integration using ε-connections. In: Stuckenschmidt, H., Parent, C., Spaccapietra, S. (eds.) Modular Ontologies. LNCS, vol. 5445, pp. 293–320. Springer, Heidelberg (2009)
10. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged Configuration Using Feature Models. In: Nord, R.L. (ed.) SPLC 2004. LNCS, vol. 3154, pp. 266–283. Springer, Heidelberg (2004)
11. Czarnecki, K., Hwan, C., Kalleberg, K.T.: Feature Models are Views on Ontologies. In: Software Product Line Conference, vol. 1 (2006)
12. d'Aquin, M., Haase, P., Rudolph, S., Euzenat, J., Zimmermann, A., Dzbor, M., Iglesias, M., Jacques, Y., Caracciolo, C., Aranda, C.B., Gomez, J.M.: NeOn Formalisms for Modularization: Syntax, Semantics, Algebra. Deliverable 1.1.3, NeOn Integrated Project (2008)
13. de Oliveira Junior, E.A., Gimenes, I.M., Huzita, E.H.M., Maldonado, J.C.: A variability management process for software product lines. In: Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research, pp. 225–241 (2005)
14. Ensan, F.: Semantic Interface-Based Modular Ontology Framework. PhD thesis, University of New Brunswick (2010)
15. Gruber, T.R.: Toward Principles for the Design of Ontologies Used for Knowledge Sharing. International Journal Human-Computer Studies 43, 907–928 (1993)
16. Kang, K., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M.: FORM: A feature-oriented reuse method with domain-specific reference architectures. Annals of Software Engineering 5(1), 143–168 (1998)
17. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report. Carnegie-Mellon University Software Engineering Institute (1990)
18. Lee, J., Muthig, D.: Feature-oriented variability management in product line engineering. Communications of the ACM - Software Product Line 49(12) (December 2006)
19. Mohan, K., Ramesh, B.: Ontology-based support for variability management in product and families. In: Proceedings of the 36th Annual Hawaii International Conference on System Sciences, p. 9. IEEE (2003)
20. Quinton, C., Haderer, N., Rouvoy, R., Duchien, L.: Towards multi-cloud configurations using feature models and ontologies. In: Proceedings of the 2013 International Workshop on Multi-Cloud Applications and Federated Clouds, MultiCloud 2013, pp. 21–26. ACM, New York (2013)
21. Rector, A., Brandt, S., Drummond, N., Horridge, M., Pulestin, C., Stevens, R.: Engineering use cases for modular development of ontologies in OWL. Applied Ontology 7, 113–132 (2012)
22. Shvaiko, P., Euzenat, J.: Ontology Matching: State of the Art and Future Challenges. IEEE Transactions on Knowledge and Data Engineering 25(1), 158–176 (2013)
23. Sinnema, M., Deelstra, S.: Classifying variability modeling techniques. Journal of Information and Software Technology 49(7) (July 2007)
24. Spaccapietra, S., Menken, M., Stuckenschmidt, H., Wache, H., Serafini, L., Tamilin, A.: D2.1.3.1 - Report on Modularization of Ontologies (July 2005)
25. Stuckenschmidt, H., Parent, C., Spaccapietra, S. (eds.): Modular Ontologies. LNCS, vol. 5445. Springer, Berlin (2009)
26. Wang, H.H., Li, Y.F., Sun, J., Zhang, H., Pan, J.: Verifying feature models using OWL. Web Semantics: Science, Services and Agents on the World Wide Web 5(2), 117–129 (2007)
27. Zaid, L.A., Kleinermann, F., De Troyer, O.: Applying semantic web technology to feature modeling. In: Proceedings of the 2009 ACM Symposium on Applied Computing, SAC 2009. ACM, New York (2009)