Change and Version Management in Variability Models for Modular Ontologies

Melanie Langermeier¹, Thomas Driessen¹, Heiner Oberkampf¹,², Peter Rosina¹ and Bernhard Bauer¹

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

Keywords: Variability Model, Modular Ontologies, Version Management, Change Management, Enterprise Architecture Management.

Abstract: Modular ontology management tries to overcome the disadvantages of large ontologies regarding reuse and performance. A possibility for the formalization of the various combinations are variability models, which originate from the software product line domain. Similar to that domain, knowledge models can then be individualized for a specific application through selection and exclusion of modules. However, the ontology repository as well as the requirements of the domain are not stable over time. A process is needed, that enables knowledge engineers and domain experts to adapt the principles of version and change management to the domain of modular ontology management. In this paper, we define the existing change scenarios and provide support for keeping the repository, the variability model and also the configurations consistent using Semantic Web technologies. The approach is presented with a use case from the enterprise architecture domain as running example.

1 INTRODUCTION

Knowledge management is an important aspect in organizations. Typically, different models are used to capture all relevant aspects of the organization. These models can extend each other, but can also overlay in some parts. Furthermore, the semantics of two models can exclude the use of both in one application. The composition of modular knowledge models to one application ontology is dealt with in the research area of modular ontology management. A specific domain ontology is created through selection of different modules from an ontology repository. To manage the complexity of the different variants that exist for the composition of an application ontology, we proposed in (Langermeier et al., 2013) the use of variability management (MOVO). This is a technique from product line engineering, that focuses on a consequent and explicit documentation of the variability on software artifacts. Such a documentation enables an individualization of software products while keeping control of the rising complexity of the variants. Variability models (VM) can also be used for modular ontologies to formalize the dependencies, that are annotated in them (e.g. owl:imports). Furthermore, these models also formalize domain independent requirements, for instance, that the domain Ontology A and Ontology B should not be used together. For the creation of a configuration, this variability knowledge can be automatically processed, and therewith supports the domain expert in creating a consistent configuration. The existing MOVO approach, however, merely presents a methodology to attain an initial setup, but does not describe how to handle changes in the VMs, their configurations and the respective ontologies. One major challenge is to keep the overall system consistent when changes occur. Our aim is to extend MOVO with concepts and techniques to support change and version management for modular ontology management. This includes concepts, to relate different versions and configurations to each other, but also technical support to fulfill consistent version updates or deletions. In section 2 we summarize the MOVO approach we want to extend. The foundations of versioning in ontology management are presented in section 3. The required change events, to support in the MOVO extension, are described in section 4 using an enterprise architecture (EA) use case as running example. The technical realization of the change management approach is described in section 5) using OWL 2 and SPARQL. We conclude with an evaluation and discussion of related work.

Langermeier M., Driessen T., Oberkampf H., Rosina P. and Bauer B.
Change and Version Management in Variability Models for Modular Ontologies.
DOI: 10.5220/0004953603830390
ISBN: 978-989-758-028-4
Copyright 2014 SCITEPRESS (Science and Technology Publications, Lda.)
2 VARIABILITY MODELS FOR MODULAR ONTOLOGY MANAGEMENT

In software product line engineering, the different variants of the product are managed through VMs. Those models allow the explicit and consequent documentation of the software artifacts’ variability in order to enable reuse in the development process (Chen et al., 2009). An overview of techniques for variability management can be found in (Chen et al., 2009) and (Sinnema and Deelstra, 2007). One well-known technique is feature modeling. Features capsule behavior, visible to the end user, on a logical layer (Beuche et al., 2004). Examples for feature modeling approaches are (Kang et al., 1990), (Kang et al., 1998) or (Czarnecki et al., 2004). Modular ontology management tries to overcome the issues of difficult reuse and performance challenges in large ontologies. Therefore, the idea is to develop small ontological models, called modules, that can be composed to a bigger application ontology. These small modules enable (partial) re-use, more efficient reasoning, easier maintenance and collaborative development (Spaccapietra et al., 2005; Stuckenschmidt et al., 2009). Using OWL 2 provides a standardized set of vocabulary to describe the modules as well as the dependencies between the single modules. These are for example the relations owl:ontologyIRI, owl:imports or version information (see section 3) (Motik et al., 2012). In (Langermeier et al., 2013) we introduced an approach for the management of the dependencies between such ontological modules as well as their composition to one application ontology. We decided to use VMs to formalize and reason about the dependencies between modular ontologies. For the creation of such a variability model, first, an ontology repository (OntRepo), including all used modular ontologies, has to be established. Then the dependencies between these modules are analyzed and formalized in the ontological variability model $V_M$. Using this model as a basis a knowledge engineer (KE) can create an integrated variability model $V_M$ through strengthening or extending $V_M$. Finally, the domain expert (DE) can select the features of $V_M$ that shall be included. Additional modules, that are necessary to get a consistent application ontology, will be added automatically. An overview of this concept with the used models and their dependencies is given in Figure 1. Figure 2 is an example of $V_M$ for the EA domain in order to enable a combined meta model from two different frameworks. It is modeled using the feature modeling approach from (Kang et al., 1990). The model captures not only the ontological restrictions in the single modules, but also restrictions from the domain, modeled by a KE. Each rectangle represents a feature. The shaded ones indicate a possible configuration for an application ontology.

3 STATE OF THE ART

Versioning in Modular Ontology Management. One of the main problems using modular ontologies is to manage changes systematically and being aware of their impacts. Ontology development is a continuous process, since ontologies evolve and have to be adapted to a changing environment and addi-
tional knowledge. A high modularity intensifies the problem, since a change in one ontology can affect various other dependent ontologies (Stuckenschmidt and Klein, 2007; Flouris et al., 2008). (Stuckenschmidt and Klein, 2007) characterize different kinds of changes in their work: (1) Changes that do affect the logical theory and therewith affect the compiled subsumption relationships and (2) changes that affect only the syntactic representation or the names of concepts and relations. Furthermore, they state that in real-world scenarios changes mostly are of the second type and therefore are harmless changes. Nevertheless, also the smaller kind of harmful changes require methods to deal with since they can have an impact on other modules. Therewith, we do not focus on the effects of changes within an ontology or the effects on other ones in this paper. Merely, we address the problem to find that places in an organization where an ontology, that has or should be changed, is in use. When doing changes to an ontology, this will always result in a new version. A versioning method is required, which will deal with the different versions of an ontology and the effects of changes to them (Klein, 2001). A method supporting ontology versions has to keep track of dependencies between them and provide access to ontologies, (prior) versions of ontologies and their respective usages (Flouris et al., 2008).

**Version Information for Ontologies.** To represent version information, OWL 2 provides different annotation properties (Motik et al., 2012). First, `owl:ontologyIRI` can be used to identify an ontology. If an ontology IRI is specified, one can additionally declare an `owl:versionIRI`. Ontologies with the same ontology IRI but different version IRIs belong to one ontology series. In a series there is exactly one current ontology which should be accessible through the ontology IRI. Using the `owl:priorVersion` property one can relate an ontology to its previous versions. Information about compatibility between versions of an ontology can be represented by `owl:backwardCompatible` or `owl:incompatibleWith`. There is no clear definition of `owl:incompatibleWith` within the specification. Obviously two ontology versions are incompatible, if they contain contradicting statements\(^1\), but one could also regard two versions as incompatible, if not all concepts of the prior version have the same semantics in the new version. General version information, like a version number, a date or other information can be added using the annotation property `owl:versionInfo`.

---

\(^1\)http://www.w3.org/TR/owl-guide/.

### 4 EXTENDING MOVO FOR VERSION AND CHANGE MANAGEMENT

The four basic functions of persistent storage, create, read, update and delete, are called CRUD in the domain of computer science. (Langermeier et al., 2013) shows how to read from the OntRepo and VMs, in the following the create, update and delete (CUD) functions are shown. In the MOVO approach introduced in section 2 four results are established: (1) an ontology repository (OntRepo), containing the relevant ontological modules for the organization, (2) an ontological variability model \(VM_O\), formalizing the dependencies between the modules, (3) an integrated variability model \(VM_I\), formalizing additional dependencies according to a specific domain under consideration of \(VM_O\) and (4) a specific configuration of modules, which is conform to \(VM_I\) and serves as application ontology. For each function, the existing dependencies between the OntRepo, the different VMs and their configurations are shown, as well as how these dependencies can be utilized to end up with a consistent overall system, after performing one of the CUD functions. Typically, such a repository stays not stable over time, it changes. To be able to manage these changes and ensure the correctness of the existing application ontologies, we enhance MOVO with concepts for the classification and connection of different VMs as well as the configurations based on those VMs. Figure 3 shows MOVO extended with concepts for versioning and tracing. To be able to trace configurations to the \(VM_I\) they are built on, a configuration of/has configuration relationship is required. To connect different \(VM_I\)s, a successor/predecessor relationship is introduced. As proposed for ontologies by the OMG, we also use backward compatible and incompatible relationships between \(VM_I\)s.
To mark obsolete $V_{M_I}$s, the label deprecated is introduced. The technical details about the new relationships are explained in section 5. In the following the different possible change events are illustrated and their effects are shown using the Enterprise Architecture use case of (Langermeier et al., 2013) as running example (see figure 2).

## 4.1 Changes in the Ontology Repository

**Create Ontology.** In order to model the distribution and communication path of the software applications in an organization, the KE wants to add an ontology module about communication and distribution. This new module has two important dependencies to existing modules in the repository. The new ontology is added to the OntRepo in order to be further processed. The $V_{M_D}$ data set is updated automatically using the setup steps described in section 5.2. This ensures that the OntRepo and $V_{M_D}$ are always consistent to each other. In the use case, the new ontology will not be included in $V_{M_I}$ at that time. It is the responsibility of the KE to decide, whether the new ontology should be included in an existing $V_{M_I}$. In that case, the changes to $V_{M_I}$ have to be published as a new version (for more details see Update $V_{M_I}$).

**Update Ontology.** When updating an ontology, only the ontology’s version information is considered and internal ones, i.e., semantic or syntactic changes are not interpreted. That means, when an updated version of an ontology is added to the OntRepo, the $V_{M_D}$ is updated differently, according to the ontology’s comprising OWL annotations ($owl$:backwardCompatibleWith and $owl$:incompatibleWith). If only an $owl$:priorVersion annotation is made, without further information about compatibility, this is treated as an incompatible update. The first and simple case is the annotation of $owl$:backwardCompatibleWith: the ontology can be simply included in the OntRepo. The $V_{M_D}$ must not be changed, since an ontology’s IRI always represents its newest version. If applicable, additional entailed dependencies of this newer version are also checked, adapted and added to $V_{M_D}$. The second case implies, that the ontology is annotated with $owl$:incompatibleWith. In this case, the representing features of the old and new version get translated to an XOR group in the feature model and both are kept in $V_{M_D}$. For example The TOGAF Business module is updated to version 2.0, while keeping it backward compatible with the prior version 1.0. The Iteraplan Business module is also updated to version 2.0, but will then be incompatible with the prior version. Figure 4 visualizes the changes in the EA $V_{M_D}$ through the update of Iteraplan Business. To keep the Iteraplan Business feature mandatory, an empty feature is created which is then decomposed using the XOR group. Afterwards, the KE has to determine if the changes made to the newer version have an impact on the already existent mappings. When necessary, they have to be adapted manually, according to the changes made to the ontology. If the KE wants to use the new versions in the $V_{M_I}$, this has to be done manually via an update (see Update $V_{M_I}$).

**Delete Ontology.** Although deletion of an existent and used ontology is very unlikely to happen, the following paragraph describes how to keep the overall system consistent after such an event. This change event is illustrated through deletion of the TOGAF Motivation module as well as the old versions of TOGAF Business and Iteraplan Business. In contrast to the former two operations the OntRepo is unaffected for the moment. The to be deleted ontology is marked as deprecated. This implies, that the $V_{M_I}$s containing that ontology are marked as deprecated too. Since the TOGAF Motivation module is not used in any $V_{M_I}$, we are able to safely remove this module from the OntRepo and $V_{M_D}$. Whereas the modules TOGAF Business and TOGAF Iteraplan are included in an $V_{M_I}$, we cannot remove these modules. To be able to execute the deletion, the KE has to delete the respective $V_{M_I}$s (see Delete $V_{M_I}$).

## 4.2 Changes in the Set of $V_{M_I}$s

**Create $V_{M_I}$**. The creation of a new $V_{M_I}$ does not have any impact, neither on the OntRepo nor on existing configurations.

**Update $V_{M_I}$**. A $V_{M_I}$ can only be updated via the creation of a new version. This new version has
to be linked via the successor-relationship to its prior one. Using the labels incompatible with and backward compatible more semantic information about the update can be made. In the new version of $VM_I$ for EA (V2) only the newer version of the Iterplan Business module should be included. Due to the incompatibility of the module versions also the two $VM_I$ versions are incompatible. Whenever an $VM_I$ is updated, all DEs of affected configurations are determined via the link between the $VM_I$ and its configurations. If the DE updates the configuration to the new version of the $VM_I$, the link between the configuration has to be updated accordingly.

Delete $VM_I$. A $VM_I$ can only be deleted, if there is no configuration using it. Following, before deleting, the $VM_I$ has to be marked as deprecated. If the DE decides to switch his application to $VM_I$ V2, there is no configuration using $VM_I$ V1 and it can thus be removed. Every time a $VM_I$ is removed by a KE, this $VM_I$ has to be checked for a deprecated ontology. If such an ontology was found, and no other $VM_I$ is using this ontology, it can be removed safely from the OntRepo and $VM_I$. When deleting $VM_I$ V1, we get informed, that now the deprecated modules TOGAF Business V1 and Iterplan Business V1 are no longer in use in any configuration or $VM_I$, and can be removed safely.

4.3 Events Concerning the Configurations

Create/Update Configuration. Every time a DE creates or updates a configuration, the link to the respective $VM_I$ (vmo:configurationOf) will be created/updated.

Delete Configuration. A configuration can be deleted by a DE at any time. If the corresponding $VM_I$ is marked as deprecated and has no more pending configurations left, this $VM_I$ can be deleted too. In our use case, after the deletion of the old configuration, we get informed that now the deprecated $VM_I$ V1 is no longer used by any configuration. We are now able to remove $VM_I$ V1.

5 REALIZATION

In the following the extended version of the Variability Model Ontology (VMO), the storage of ontology versions in the OntRepo, the representation of them as features in the ontological variability model and the change management are described.

5.1 Variability Model Ontology

The initial version of the VMO is described in (Langermeier et al., 2013). The main classes of the $VMO_0$ are vmo:Feature 2, owl:Composition, vmo:Alternative_Composition and vmo:Or_Composition. We added several new object properties to the $VMO_0$ (see figure 5). For instance,

we defined an object property vmo:isRealizedIn to relate features with ontology IRIs of the OntRepo (see Figure 8). Domain and Range of this property is vmo:Feature and owl:Ontology respectively. Further, we added a class vmo:VMI for the integrated Variability Models ($VMI_S$). A $VMI_S$ is an instance of vmo:VMI and is represented through a named graph, containing features and relations between them. Named graphs are a good way to group triples in a triple store and further the graph URI (i.e. the URI of the $VMI_S$ instance) can be used to relate the $VMI_S$ instance to its dependent configurations: A configuration is a valid selection of features from one $VMI$. Since each feature represents an ontology, a configuration represents indirectly also a set of ontologies. The two properties vmo:configurationOf and vmo:hasConfiguration are used to connect a configuration with the corresponding $VMI$. Selected features are related to the configuration with the property vmo:selectedFeature. The object property vmo:isRealizedIn is used to relate vmo:Features with the respective ontology the feature is representing. Additionally a transitive property vmo:successor (with inverse vmo:predecessor) is used to interconnect different versions of $VMI_S$ in the right order. Domain and range of both properties is vmo:VMI. Different versions of $VMI_S$ are backward compatible (and linked with vmo:backwardCompatibleWith property).

$\text{Figure 5: Object properties of the Variability Model Ontology (VMO). New properties are in bold.}$

We write vmo:Feature instead of <http://www.dslab.org/ontologies/movo-ontology.owl#Feature>.
if all configurations created by the older version can still be created using the newer version. Otherwise they are linked with `vmo:incompatibleWith`. The property `owl:deprecated` is generally used for marking IRIs as deprecated. We use this annotation property to flag outdated and thereby obsolete ontologies, which comes into play when ontologies shall be deleted. A similar property (`vmo:deprecated`) is used to flag `VMIs`, configurations or features as as deprecated.

5.2 Set-up of OntRepo and VMO

For the technical realization of our approach we set up an Apache Jena Fuseki triple store with two data sets (see figure 6). The first data set (ontrepo) is

![Figure 6: The triple store contains two data sets. One for the ontology repository and another for the variability models. A SPARQL Query Repository is used to access and modify data.](image)

the ontology repository, the second (vmo) is used to store the Variability Model Ontology, the features with their dependencies and the `VMIs`. Since the ontology repository contains several versions of one ontology we separate them using named graphs. This is necessary since concepts of different versions share the same namespace. The named graphs reuse the version IRIs of the ontology versions or the ontology IRI if no version IRI is defined. Ontology annotation properties such as version IRIs or import statements are additionally stored in the default graph. These relations are shown in Figure 7. In the next step, we automatically create features in the vmo dataset for all ontology IRIs in the default graph of the OntRepo data set. This is done using SPARQL UPDATE queries from the query repository. If there are incompatible versions of one ontology, we additionally create one `vmo:Alternative_Composition` and relate it to the version IRIs and the ontology IRI through `vmo:consistsOfFeatures` property. Since `vmo:consistsOfFeatures` has range `vmo:Feature` these versions become also `vmo:Feature` in the vmo data set. An example of an alternative composition with relations to the IRIs is shown in figure 8. Within

![Figure 7: Relations between ontology IRIs in the OntRepo default graph.](image)

the creation process of `VMi` the KE can then decide whether to use the feature for the ontology (which will result in the use of the most current version) or the composition rule in order to allow the use of one of the different versions.

5.3 Changes in the Ontology Repository

If a new ontology is added to the ontology repository, the initial set-up steps for integration into OntRepo and `VMi` as described in section 5.2 are repeated. If a new version of an existing ontology is added to the repository, we distinguish two cases: If the new version is backward compatible, we simply repeat the set-up steps. Since the corresponding feature points to the ontology IRI and not to the version IRI, we do not have to change the `VMi`. However configurations using the respective features have to be recreated. If the new version is incompatible with the prior version we need to create a new alternative composition in `VMi` or adapt an existing one in adding a new feature for the new version. If one ontology should be deleted from the ontology repository, first the IRI of the version or ontology has to be flagged as deprecated using the `owl:deprecated` property as mentioned above. Features in the `VMi` dataset, which are related to a deprecated ontology via the `vmo:isRealizedIn` property, as well as corresponding `VMIs` are set as deprecated using `vmo:deprecated` property too. The feature and corresponding ontology are deleted after the last
5.4 Changes in VM₁

A backward compatible new version of a VM₁ simply obtains all references from configurations to the old version. The named graph of the old VM₁ is dropped afterwards. Both operations are automatically performed using SPARQL UPDATE queries. Configurations are not linked to the new VM₁, if it is incompatible to the former version. In the case of ontology updates in the OntRepo all VM₁s containing features with references to the respective ontology need to be updated.

6 EVALUATION

Comparing our versioning method with the issues stated in section 3, we support most of them. We are able to manage different versions of the same ontology in one data set and also manage the dependencies between those versions. We consider the dependencies between different ontologies as well as their usage in specific applications. Therewith, our method ensures, that the information about which ontology version is used in which application is always correct and hinders an automatic update to newer ontology versions that may cause inconsistencies. Older versions of an ontology are kept in the data set while marking them as deprecated. If no other ontology or application is using them, they can be automatically removed. Additionally, we evaluated the method in a use case (see running example in section 4). Therefore, we reused the established VM₁ (Figure 2) from (Langermeier et al., 2013) and executed change operations. This VM is created for the Enterprise Architecture (EA) domain in order to be able to combine different meta models and mix and match them according to the requirements of a specific organization. The different versions of the models (current, deprecated, deleted) with their dependencies, that exist after executing all the change operations, are illustrated in figure 9.³

7 RELATED WORK

Ontology Evolution. As stated in (Stojanovic et al., 2002), most of the research in the ontology engineering field is done on construction issues. However, evolution of ontologies requires different approaches. Our work was focused on the evolution of multiple, modular ontologies on an abstract feature level, whereas most of the related work copes with single ontologies and their evolution over time. Examples can be found manifold, like in (Stojanovic et al., 2002). In this work, the authors present a 6-phase methodology in order to keep an ontology consistent over time. The key step of this methodology is to differentiate between syntactic and semantic changes. Another approach is presented by (Plessers et al., 2007), where the authors created an own framework for detecting changes, made to OWL DL ontologies. Using a predefined set of change definitions, they are able to provide different overviews of changes made to an ontology, depending on the user’s role.

Ontology Repositories. Ontology repositories store either domain specific or independent ontologies, provide users with browse and search interfaces, and can offer other functionality like reviewing, annotating or editing and mapping ontologies. The ontologies’ metadata, for instance, author, version, date or scope, is a valuable instrument for these functionalities. The mapping metadata can even be used to relate concepts from different ontologies (Noy et al., 2008). Generally, there are two different types of ontology repositories: the ones that are automatically crawled (Swoogle, Watson etc.) and the ones where the ontologies are submitted manually by users (Noy et al., 2008). The initialization and extension of our proposed repository is a manual effort. It is still rudimentary and does not provide a user interface. Nonethe-

³The test set for the evaluation with the data sets, queries, scripts and a documentation is published in http://megastore.uni-augsburg.de/get/HAh0VXS7qw/. The test set was built upon the results from (Langermeier et al., 2013).
less, we use the meta data provided with the ontologies in order to find mapping and version information.

8 CONCLUSION

In this paper we proposed a method for change and version management for MOVO. By reusing the well known CRUD functions, we strongly reduced the complexity for change management. We show the impact of those events for the OntRepo, its VM_O, the different VMs and their configurations. Using technologies from the Semantic Web technology stack, we implemented the method to calculate the impacts automatically. Some parts still require the decision of a KE or a DE. Applying this method we enable MOVO to be used by long-term systems without losing manageability due to increasing complexity. Future work has to be done to enable our approach handling the changes made within an ontology and especially consider the semantic impacts these modifications have on other ontologies. We also want to research more expressive approaches for defining relations between ontologies, e.g., E-Connections (Cuenca Grau et al., 2009), Package Based Description Logics (P-DL) (Bao et al., 2006), Distributed Description Logics (DLL) (Borgida and Serafini, 2003) or the Interface-based modular ontology Formalism (IBF) (Ensaf, 2010). Finally a prototype tool implementation has to be done with a first user interface.

REFERENCES


