Semantic Web – Interoperability, Usability, Applicability 0 (2010) 1
IOS Press

KOMMA: An Application Framework for Ontology-based Software Systems

Ken Wenzel *
Fraunhofer-Institute for Machine Tools and Forming Technology IWU,
Department for Production Planning and Ressource Management,
Reichenhainer Str. 88, 09126 Chemnitz, Germany

Abstract. The Knowledge Modeling and Management Architecture (KOMMA) is an application framework for Java software systems based on Semantic Web technologies. KOMMA supports all application layers by providing solutions for persistence with object triple mapping, management of ontologies using named graphs as well as domain-specific visualization and editing of RDF data. This article gives a short introduction to KOMMA's architecture and illustrates its usage with examples.

Keywords: application framework, eclipse, rdf, semantic web, object triple mapping, model-driven architecture

1. Introduction

Semantic Web technologies have gained popularity in the fields of information integration and semantic search. Therefore research and development activities mainly focused on ontologies, ontology editors, storage and inference solutions for RDF data as well as on generic solutions for presenting, navigating and editing RDF data sets.

Among other things, these activities led to the development of

- large upper ontologies including SUMO [1] or domain-specific ones such as OntoCAPE [2], the ISO 15926 ontologies [3,4] or the Open Biomedical Ontologies [5],
- ontology engineering software like Protégé [6], TopBraid Composer [7] or NeOn Toolkit [8],
- RDF frameworks like Jena [9] and OpenRDF Sesame [10] as well as
- semantic wikis like OntoWiki [11] or Semantic Media Wiki [12].

Ontology engineering software mainly focuses on the development of domain ontologies for their later usage in ontology-based software systems. This results in a strong orientation towards graphical presentation and editing support for ontology languages, rather than for data described by domain ontologies.

Another approach is followed by semantic wikis. These systems usually provide form-based access to large RDF data sets, as implemented by OntoWiki, or, as in the case of Semantic Media Wiki, offer RDF-based tagging and searching capabilities for existing hyperlinked data sets.

We argue that the wide adoption of Semantic Web technologies can be accelerated by providing sophisticated application frameworks that lower the barriers for the development of RDF and OWL-based software systems.

These frameworks should reduce the effort for transformations of model definitions into implementations of software systems. One approach that fulfils this requirement is Model-driven engineering (MDE), a software development methodology that simplifies the creation of model-based applications by providing tools that support the required transformation tasks.

Because knowledge representation languages like RDFS or OWL can also be regarded as some type of modeling language in the sense of MDE, we've started the development of KOMMA by investigating several MDE tools.

*Corresponding author. E-mail: ken.wenzel@iwu.fraunhofer.de

One of these is the Eclipse Modeling Framework (EMF) [13]. It combines a modeling language for structured data models with an application framework and a code generation facility to transform models into classes for the Java programming language.

The EMF project shows that there is large industrial request for model-based application frameworks. Over recent years this project was able to acquire a large community that actively uses and develops a variety of solutions around the EMF core components.

EMF's modeling language Ecore is a subset of the Meta Object Facility (MOF), an extensible model-driven integration framework, defined by the Object Management Group (OMG). The application framework delivered with EMF applies the adapter pattern [14] to provide tailored presentation and editing capabilities for different model elements. Code generation is used to create classes for domain objects as well as to implement generic adapters for presentation and editing.

The goal of KOMMA is to combine OpenRDF Sesame as an RDF persistence provider with a sophisticated solution for object triple mapping and the flexible visualization and editing capabilities of EMF into a unified application framework for RDF- and ontology-based software systems.

2. Related Work

This section covers some related projects that influenced the development of KOMMA or have similar goals.

2.1. Empire

Empire [15] is an implementation of the Java Persistence API (JPA) [16] for RDF and provides a solution for object triple mapping based on JPA's ORM principles. Empire supports querying of RDF databases with SPARQL or Sesame SeRQL. The mapping of Java classes to RDF types is realized by using standard JPA annotations which are extended by some RDF specific ones for namespaces, RDFS classes and RDF properties.

Empire aims at replacing existing JPA providers for relational databases to simplify the transition of legacy systems towards RDF-based data models.

2.2. OpenRDF Elmo and AliBaba

Elmo provides a JPA compatible implementation of an entity manager that provides object triple mapping for Sesame repositories. In contrast to Empire, Elmo uses its own annotations for the mapping of Java classes to RDF types and does not rely on annotations defined by JPA.

AliBaba is developed as a successor to Elmo and uses similar principles for object triple mapping. It is more tightly coupled with Sesame and does not adhere to the JPA specification.

AliBaba provides RESTful subject-oriented implementations of client and server libraries for distributed storage of documents and RDF metadata. One of its most notable features is the extensibility mechanism through a messaging ontology that enables the implementation of domain logic by embedding Java or Groovy code into RDF data. This mechanism in combination with AliBaba's RESTful services allows for creation of flexible RDF-based applications.

KOMMA reuses parts of Alibaba's solution for object triple mapping that is also exposed through Alibaba's client interface.

2.3. Callimachus

Callimachus [18] is a Semantic Web framework that aims at the development of hyperlinked web applications. It is largely based on OpenRDF AliBaba and applies its extension mechanisms to provide a pluggable architecture for web applications. Callimachus uses a template engine in combination with RDFa to generate forms for viewing and editing of resources.

2.4. EMFTriple and EMF4SW

EMFTriple [17] provides basic object triple mapping for EMF by partially implementing JPA. Its main goal is to provide RDF persistence for arbitrary EMF models which is realized by synchronizing RDF data sources with corresponding EMF models.

This approach enables the usage of existing EMF technologies, like the editing framework or model transformations and comparisons, for RDF data. Using EMF to represent RDF data implies that the associated data model and hence the ontologies need to be expressed in Ecore which is less expressive than OWL.

EMF4SW [17] provides EMF Ecore models for RDF and OWL. These models are comparable to ODM which is developed by OMG to create an MOF-based

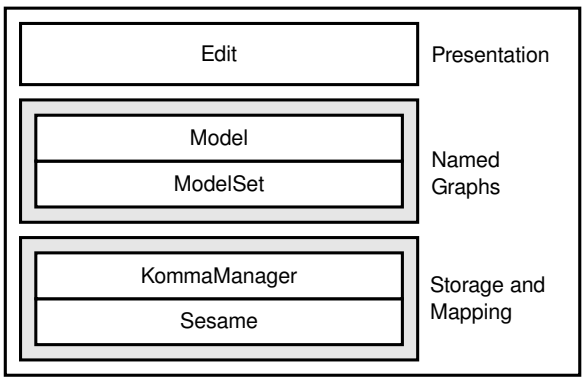

Fig. 1. Architecture overview.

meta model for the definition of ontologies. EMF4SW can help create tools for ontology engineering but is less useful for domain-specific ontology-based applications.

3. Architecture and Object Model

KOMMA uses a layered architecture as depicted in Figure 1 to improve the reusability of its components.

The storage and mapping layer (Section 4) is based on OpenRDF Sesame for managing RDF data and provides an advanced solution for the mapping between Java objects and RDF triples. It is not aware of OWL or other ontology markup languages and hence does not consider any ontological information contained in the data. This ensures its reusability for all kinds of RDF based systems no matter which kind of ontology language, if any, is used.

The next layer on the stack deals with RDF named graphs (Section 5). It is aware of OWL ontologies with their dependencies and uses models and sets of models to manage them. In its current implementation the data of each OWL ontology is mapped to one model that itself is contained in a set of interrelated models. Consequently, a model set contains models that constitute the imports closure of all contained ontologies.

The last layer implements mechanisms for the presentation and editing of RDF resources (Section 6). It uses the adapter pattern to create views and editors for RDF resources and already provides many generic implementations and base classes to quickly create editable lists, tables or tree views.

Figure 2 shows KOMMA's core object model. All objects representing RDF nodes implement the interface `IReference`. In the case of named nodes an `IReference` is associated to a corresponding `URI`, for blank nodes this `URI` is `null`.

By defining both interfaces, `URI` and `IEntity`, as specialization of `IReference`, we get a normalized class hierarchy where objects of type `IEntity` can be used interchangeably with basic references (`URI` for named nodes, `IRefence` for blank nodes) to RDF nodes.

The interface `ILiteral` complements the interface `IReference` to represent RDF literals.

RDF resources mapped to Java objects are managed by an instance of `IKommaManager` – KOMMA's equivalent for RDF to JPA's entity manager for relational databases. This `IKommaManager` is responsible for object triple mapping which is described in Section 4.

`IResource` is a subject-oriented interface for reflective access to properties of RDF nodes. Among other things, it can be used for determining the direct RDF types of a resource or for adding and removing of property values.

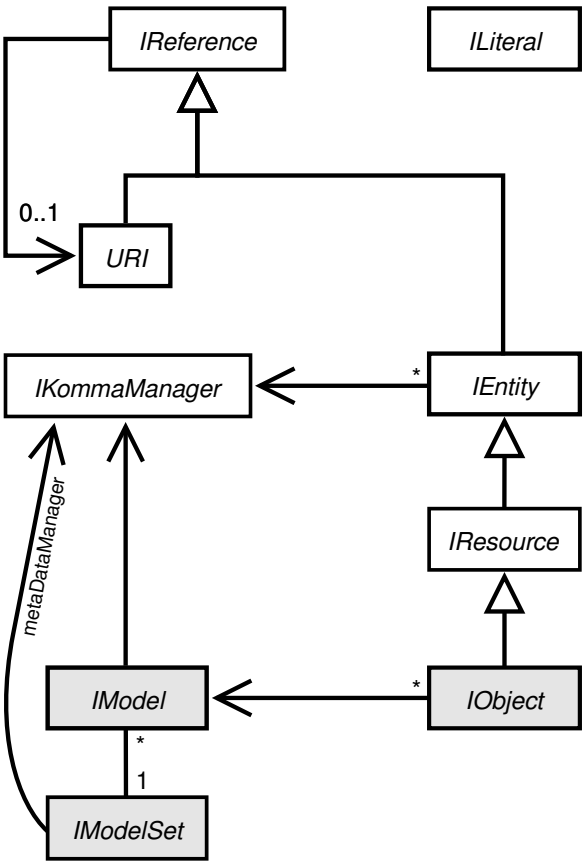

Fig. 2. Core object model.

The shaded classes in Figure 2 are core components of KOMMA's subsystem for the management of RDF named graphs and OWL ontologies which is covered by Section 5.

4. Object Triple Mapping

KOMMA's solution for object triple mapping is based on the work done for the OpenRDF [10] projects Elmo and AliBaba.

4.1. Mapping of RDF types to Java classes

RDF resources may have multiple associated types while Java supports only single inheritance and hence only objects of one type. This fact imposes problems for the mapping of RDF resources to Java objects. One solution to circumvent this issue is by using the adapter pattern to create multiple views for a Java object that reflect the different types of the underlying RDF resource. For example, Jena uses the adapter pattern to create different views for RDF nodes to reflect different semantics.

KOMMA follows another approach by using an engine that is able to combine multiple interfaces and classes encapsulating object behaviour into one class that represents the union of multiple RDF types. These classes are generated dynamically by KOMMA at runtime of the application.

Interfaces and classes can be mapped to RDF types by using the `@iri` annotation that expects an absolute IRI to associate its target with an RDF resource.

When KOMMA needs to create a new Java class to represent an RDF resource as Java object, it first retrieves its RDF types and then determines all associated Java interfaces and classes. The resulting set of interfaces and classes is combined into one class that embodies the resource's types.

The example in Figure 3 shows two interfaces for OWL properties that are associated to their corresponding RDF types `owl:SymmetricProperty` and `owl:TransitiveProperty`.

If KOMMA encounters an OWL property that is both transitive and symmetric then it also assigns both interfaces to this property.

A more complex example is depicted in Figure 5. It contains mappings for the class `Pet` and its subtypes `Cat` and `Dog`. Each pet has the ability to `yell`. Since cats and dogs usually make different sounds, the method `yell` is implemented for cats by `CatSupport` and also for dogs by `DogSupport`.

```
@iri("http://www.w3.org/2002/07/owl#
    SymmetricProperty")
public interface SymmetricProperty extends
    ObjectProperty {}

@iri("http://www.w3.org/2002/07/owl#
    TransitiveProperty")
public interface TransitiveProperty extends
    ObjectProperty {}
```

Fig. 3. Example for mapping of symmetric and transitive OWL properties.

An RDF resource of types `Cat` and `Dog` would be represented by the exemplary Java class depicted in Figure 4. The implementation of `yell` illustrates KOMMA's method chaining that calls both methods of `DogSupport` and `CatSupport`. The order of chained method calls can be defined by establishing a hierarchy between behaviour classes by using the annotation `@precedes` .

```
public class CatDogEntity
  implements Cat, Dog
{
  private IReference nodeRef;

  private CatBehaviour cat;
  private DogBehaviour dog;
  private CatSupportBehaviour catSupport;
  private DogSupportBehaviour dogSupport;

  public CatDogEntity(IReference nodeRef) {
    this.nodeRef = nodeRef;
  }

  public void yell() {
    getDogSupport().yell();
    getCatSupport().yell();
  }

  public void getCatSupport() {
    if (catSupport == null) {
      catSupport =
        new CatSupportBehaviour(this);
    }
    return catSupport;
  }
  ...
}
```

Fig. 4. Entity class generated by KOMMA's object triple mapper.

4.2. Mapping of RDF properties to object methods

The example in Figure 5 contains mappings for the properties `name` and `parent` of the type `Pet`. Both mappings are defined by annotating the corresponding getter methods with an `@iri` annotation. If a setter method is also present then the property is treated as writable, else as read-only.

The mapping of properties is implemented by using `PropertySet` objects that encapsulate the required logic to convert between their RDF and Java representation. KOMMA contains a bytecode generator to implement all behaviour interface methods that are annotated with `@iri`. The generator uses a pluggable factory for property sets to enable the usage of various storage back ends.

```
@iri("http://example.org/pets#Pet")
public interface Pet {
  @iri("http://example.org/pets#name")
  String getName();
  void setName(String name);

  @iri("http://example.org/pets#parent")
  Set<Pet> getParents();
  void setParents(Set<Pet> parents);

  void yell();
}

@iri("http://example.org/pets#Cat")
public interface Cat extends Pet {}

@iri("http://example.org/pets#Dog")
public interface Dog extends Pet {}

abstract public class CatSupport implements
    Cat {
  public void yell() {
    System.out.println(getName() + " says
        MEOW");
  }
}

@precedes(CatSupport.class)
abstract public class DogSupport implements
    Dog {
  public void yell() {
    System.out.println(getName() + " says
        WUFF WUFF");
  }
}
```

Fig. 5. Example for mapping of multiple types and object behaviours.

4.3. Scalability

As with all object-relational or object triple mapping solutions scalability is an issue. The reason is that the creation of proxy classes for mapped objects is costly and that the lazy loading of property values requires many individual database requests. KOMMA improves the application performance by using a second-level cache for mapped objects and by providing mechanisms to prefetch object properties.

4.3.1. Second-level cache

The second-level cache is realized by JBoss Infinispan [19], a data grid platform with distributed cache capabilities that also includes a tree-structured cache API.

KOMMA caches mapped objects by using a tree-like structure for each object and its associated property values.

Automatic caching can be enabled for arbitrary behaviour methods by using the `@Cacheable` annotation. An optional `key` can be provided to use prefetching as described in Section 4.3.2.

```
public interface IClass extends ... {
  @Cacheable(key =
    "urn:komma:directSuper")
  Set<IClass> getDirectSuperClasses();
}
```

KOMMA's cache invalidation policy uses Sesame's notification API to invalidate cached objects if some of their associated properties change. A problem arises if reasoning is used, since most triple stores with Sesame API do not offer notification support for changes of inferred statements. Therefore cache invalidation is also possible by calling `IKommaManager.refresh` for a corresponding entity object.

4.3.2. Prefetching of property values

KOMMA supports prefetching for mapped properties annotated with `@iri` or for return values of behaviour methods that carry a `@Cacheable(key)` annotation.

Prefetching works by using SPARQL `CONSTRUCT` queries to retrieve the corresponding property values in addition to the original results. Figure 6 illustrates an example query for retrieving classes along with prefetching of their direct super classes and RDF types.

The construct template

```
?class a <urn:komma:Result>
```

```
CONSTRUCT {
  ?class a <urn:komma:Result> .
  ?class <urn:komma:directSuper> ?super .
  ?class a ?type
} WHERE {
  ?class a rdfs:Class .
  OPTIONAL {
    ?class rdfs:subClassOf ?super .
    OPTIONAL {
      ?subClass rdfs:subClassOf ?otherSuper .
      ?otherSuper rdfs:subClassOf ?super .
      FILTER (?super != ?otherSuper)
    }
    FILTER (! bound(?otherSuper))
  }
  OPTIONAL {
    ?class a ?type
  }
}
```

Fig. 6. Example query to prefetch direct super classes and RDF types.

marks the original results while the other two templates

```
?class <urn:komma:directSuper> ?super .
?class a ?type
```

are used to prefetch the corresponding property values.

KOMMA provides a SPARQL builder that simplifies the usage of prefetching by automatically transforming `SELECT` queries into equivalent `CONSTRUCT` queries while adding constructs to preload property values.

4.4. Transactions and Concurrency

In its current implementation KOMMA leaves the handling of transactions to the underlying RDF repository. Sesame does not implement nested transactions, so neither does KOMMA.

The class `IKommaManager` contains a method `getTransaction` that returns a transaction object that can be used to `begin`, `commit` or `rollback` transactions.

Unlike most JPA providers, KOMMA does not directly implement any methods for concurrency control. This means that there is no simple way to detect concurrent changes by multiple users and transactions to the same RDF resources. This shortcoming can be circumvented by using Sesame providers like AliBaba's Optimistic Repository that implements optimistic concurrency control (OCC) or Bigdata RDF [21] that implements multiversion concurrency control (MVCC).

5. Named graphs and models

RDF named graphs are a means to structure RDF data sets. With named graphs it becomes possible to assign an IRI to a related set of RDF statements that can then be directly selected by its name within SPARQL queries.

KOMMA provides models and model sets to manage data contained in RDF named graphs.

Models implement the interface `IModel` that provides methods for loading the model's contents into a repository, for saving them to an external resource and for deleting them from a repository. A model does also provide methods to access information about modification state or about diagnostics like warnings and errors.

Each model keeps track of its imported models by using the *owl:imports* directive. The imports closure and hence the corresponding named graphs are also accessible through the model's `IKommaManager`.

Loading and saving models is done with the help of a so-called URI converter that uses a set of rules for resolving model URIs (and hence ontologies) to physical locations in the local file system or on a remote server. After resolving the correct location, a specialized URI handler for the encoded scheme (e.g. http, ftp or file) is responsible for accessing the corresponding resource.

An `IModelSet` is a group of multiple interrelated models that is backed by a Sesame repository where each model is stored in its own context (named graph).

Figure 2 reveals that each `IModelSet` has an associated `metaDataManager`. The reason is that KOMMA implements models and model sets by using its own object triple mapping mechanism. This results in the ability to handle these objects in the same way as all other RDF resources and enables the usage of SPARQL to access metadata about models and model sets.

The code in Figure 7 exemplifies the basic usage of KOMMA's core components. It demonstrates the usage of `KommaModules` to register concepts and behaviours for object triple mapping, the construction of model sets by using the `ModelSetFactory`, the basic interaction with `IKommaManager` as well as the serialization of model data to file.

```
// create module with model concepts
// and behaviours
KommaModule module = ModelCore
  .createModelSetModule(
    getClass().getClassLoader());
// use factory to create model set
IModelSet modelSet = new ModelSetFactory(
    module,
    URIImpl.createURI(MODELS.NAMESPACE +
      "MemoryModelSet"))
  .createModelSet();
// register concepts and behaviours
module = modelSet.getModule();
module.addConcept(Cat.class);
module.addConcept(Dog.class);
module.addBehaviour(CatSupport.class);
module.addBehaviour(DogSupport.class);

// create an example model
IModel model = modelSet.createModel(URIImpl
  .createURI("file:///tmp/example.owl"));
// import pets ontology
model.addImport(URIImpl.createURI(
  "http://example.org/pets"), "pets");

IKommaManager m = model.getManager();
// create famous CatDog
Pet catDog = m.createNamed(
  model.getURI().appendFragment("TheCatDog"),
  Cat.class, Dog.class);
catDog.setName("CatDog");
// create CatDog's parents
catDog.getParents().add(m.create(Cat.class));
catDog.getParents().add(m.create(Dog.class));
// let CatDog say something, output is:
//   CatDog says WUFF WUFF
//   CatDog says MEOW
catDog.yell();

// save model contents to file
Map<Object, Object> saveOptions =
  new HashMap<Object, Object>();
model.save(saveOptions);
```

Fig. 7. Example for interaction with models and model sets.

6. Editing Framework

KOMMA includes an Eclipse framework comprising generic reusable classes for building editors for RDF data models. This framework is strongly modeled after EMF.Edit, the editing framework for EMF.

Likewise, it simplifies the creation of GUI editors by providing

- content and label providers for JFace viewers,
- a command framework, including a set of generic commands for common editing operations as well as
- automatic undo and redo support for changes to models.

6.1. Content providers

Eclipse JFace is an MVC [20] framework that provides viewer implementations for displaying structured models as lists, tables or trees. It decouples the presentation of domain objects by using content and label providers to transform the underlying models into GUI elements.

For example, a JFace content provider for tree viewers has the following structure:

```
interface ITreeContentProvider ... {
  Object[] getChildren(Object parent);
  Object getParent(Object element);
  boolean hasChildren(Object element);
  ...
}
```

By implementing this interface, it is possible to transform any domain model into a tree-like representation.

6.2. Providing content and labels for RDF resources

KOMMA contains generic classes for content and label providers that use the adapter pattern to create representations for RDF resources.

The `AdapterFactoryContentProvider` implements interfaces for several JFace content providers by delegating to specialized adapters that know how to navigate the model objects. For example, an adapter for tree structures would implement the KOMMA interface `ITreeItemContentProvider` to provide items for tree viewers.

```
interface ITreeItemContentProvider ... {
  boolean hasChildren(Object element);
  Collection<?> getChildren(Object parent);
  Object getParent(Object element);
  ...
}
```

As can be easily seen, this interface has similar methods to `ITreeContentProvider` since it is directly used by KOMMA's generic content provider to implement the corresponding methods.

The `getChildren` method is, for example, implemented as follows:

```
public Object[] getChildren(Object object) {
  ITreeItemContentProvider
    treeItemContentProvider =
    (ITreeItemContentProvider) adapterFactory
      .adapt(object,
        ITreeItemContentProvider.class);
  return treeItemContentProvider != null ?
    treeItemContentProvider
      .getChildren(object) :
    Collections.emptyList()).toArray();
}
```

The `AdapterFactoryLabelProvider` works in the same way to provide labels for RDF resources. Figure 8 shows an example where the tree on the left side as well as the labels and icons are created by adapters.

6.3. Adapter factories

As stated before, KOMMA's generic content and label providers use adapters to handle different model elements. These adapters are created by an adapter factory:

```
public interface IAdapterFactory {
  Object adapt(Object object, Object type);
  boolean isFactoryForType(Object type);
}
```

An adapter factory may support different adapter types (`isFactoryForType`). In contrast to EMF where each model object keeps track of its associated adapters, KOMMA transfers this obligation to the adapter factories. This decision was made to decouple the domain model from transient applications objects. The reason is that KOMMA's domain objects are rather lightweight references to RDF resources and may be instantiated multiple times while EMF's domain objects usually exist only once in memory.

6.4. Commands

The modification of model objects is done by commands implementing the interface `ICommand`. Each command is responsible to execute or, if possible, undo and redo the encapsulated operations. The execution of commands is managed by an `ICommandStack` that also keeps track of an execution history. This command stack is associated to a model set by an `IEditingDomain` object that is also responsible for creating generic commands by using an adapter factory. Hence an editing domain encapsulates the required objects to modify one ore more models during a user session.

KOMMA simplifies the implementation of editors by providing generic command implementations for common operations like create, add, delete, move, copy or drag and drop.

The basic commands can be overridden by domain-specific adapters that provide tailored commands for the corresponding model objects. An example can be seen in Figure 8 where menu actions for the creation of new children or siblings are determined by using an adapter that is responsible for the selected element.

6.5. Automatic undo and redo

Although basic support for undoing and redoing operations is provided by the interfaces `ICommand` and `ICommandStack` the commands need to remember changes made to model elements during their execution to restore the original state in case of an undo.

KOMMA simplifies the implementation of undo and redo by a `RecordingWrapperCommand` that is able to capture all model changes during command execution. This special command uses the notification capabilities of Sesame's `NotifyingRepository` to track added and removed triples.

Undoing the command simply deletes all added triples and inserts all removed triples while redoing the command does the opposite.

7. Basic vocabulary for applications

KOMMA defines a basic ontology for applications that want to use KOMMA's generic content providers or commands. This ontology is mainly intended to specify properties that express containment relationships and to provide concepts that enable the direct mapping of EMF's Ecore models to OWL ontologies.

The KOMMA ontology defines the following containment properties:

```
komma:containsTransitive
    < komma:contains
        < komma:orderedContains
```

The property `komma:containsTransitive` is of type `owl:TransitiveProperty` to allow recursive inference of containments.

Its subproperty `komma:contains` is declared as `owl:InverseFunctionalProperty` to specify that an object may only be part of one parent object.

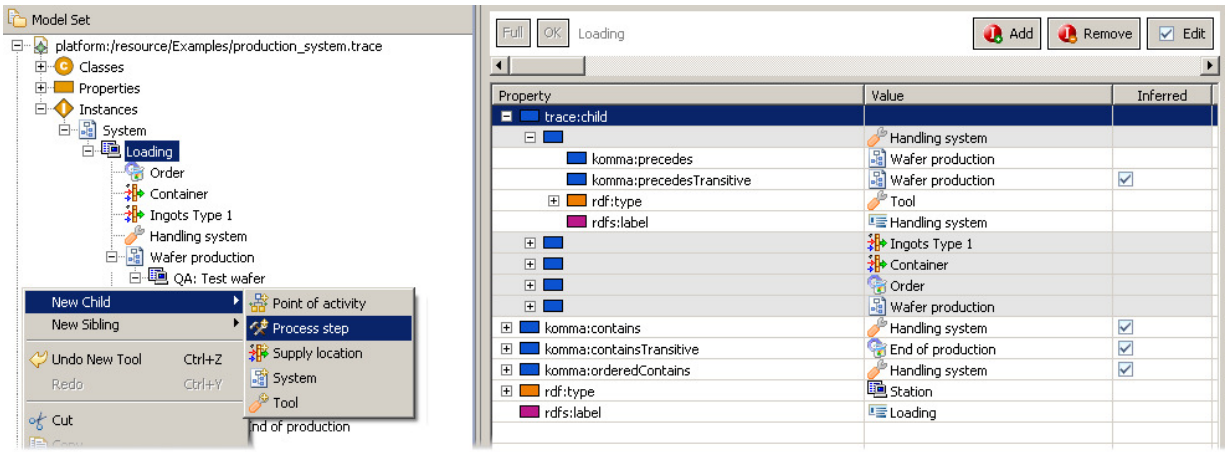

Fig. 8. Editing framework in action.

Many application scenarios require tree structures to be presented in a deterministic way where the order of elements is defined by a user. This is accomplished by using the property `komma:orderedContains` in combination with the property `komma:precedes` that defines a partial ordering between elements.

These properties are directly supported by object triple mapping through providing a specialized property set for `komma:orderedContains` that uses the property `komma:precedes` to preserve the order of its values.

The KOMMA ontology additionally introduces the concepts `Map`, `MapEntry` and some specializations with their related properties `entry`, `key` and `value` to simplify the transformation of EMF models that use the data type `java.util.Map`.

8. Contributing to KOMMA

KOMMA is offered as open source software, licensed under the terms of the Eclipse Public License (EPL) [22]. This decision was motivated by the fact that open source projects always benefit from communities contributing to their development.

The public community site for KOMMA is hosted at `komma.sourceforge.net`. This is the entry point to public forums, a bug tracker as well as KOMMA's source code repositories. The latter are based on GIT to simplify distributed development.

Source code contributions are managed by our Gerrit instance at `git.iwu.fraunhofer.de`. This code review system enables any user to participate in KOMMA's development by submitting patches for bugs or features.

9. Conclusions and future work

KOMMA complements existing model-driven application frameworks by providing an integrated architecture for RDF and ontology-based software systems. Therefore, it is able to reduce the effort for the creation of semantic desktop and internet applications.

Future developments of KOMMA include the extension of its editing framework towards more advanced domain-specific graphical and textual presentations of RDF data as well as seamless support of rich client and rich internet applications. We hope that these activities will improve KOMMA's applicability to a wide range of real-world usage scenarios.

KOMMA already implements required core features to support innovative software design paradigms like *Data, Context and Interaction (DCI)* [23]. Further research could investigate how KOMMA's role-based object triple mapping in combination with its notion of models can be used to implement context-based policies for user interaction to accomplish goals of the DCI principle.

Acknowledgements

The work presented in this paper is co-funded by the European Union with the European Fund for Regional Development (EFRE) and by the Free State of Saxony.

References

[1] A. Pease, I. Niles, and J. Li, *The Suggested Upper Merged Ontology: A Large Ontology for the Semantic Web and its Appli-*

cations, in: Working Notes of the AAAI-2002 Workshop on Ontologies and the Semantic Web, 2002.

[2] J. Morbach, A. Wiesner, and W. Marquardt, *OntoCAPE – A (re)usable ontology for computer-aided process engineering*, in: Computers & Chemical Engineering, Vol. 33, 2009, pp. 1546–1556.

[3] R. Batres, M. West, D. Leal, D. Price, K. Masaki, Y. Shimada, T. Fuchino, and Y. Naka, *An upper ontology based on ISO 15926*, in: Computers & Chemical Engineering, Vol. 31, May 2007, pp. 519–534.

[4] ISO 15926-2, *ISO-15926:2003 Integration of lifecycle data for process plants including oil and gas production facilities: Part 2. Data model.*, 2003.

[5] B. Smith, M. Ashburner, C. Rosse, J. Bard, W. Bug, W. Ceusters et al., *The OBO Foundry: coordinated evolution of ontologies to support biomedical data integration*, in: Nature Biotechnology, Vol. 25, 2007, pp. 1251–1255.

[6] The Protégé Ontology Editor and Knowledge Acquisition System, http://protege.stanford.edu/.

[7] TopQuadrant TopBraid Composer, http://www.topquadrant.com/.

[8] NeOn Toolkit, http://neon-toolkit.org/.

[9] Jena, http://jena.sourceforge.net/

[10] OpenRDF, http://www.openrdf.org/.

[11] N. Heino, S. Dietzold, M. Martin, and S. Auer, *Developing Semantic Web Applications with the OntoWiki Framework*, in: Networked Knowledge - Networked Media, S. Schaffert et al. (eds.), Springer, Berlin / Heidelberg, 2009, SCI 221, pp. 61–77.

[12] M. Krötzsch, D. Vrandecic, M. Völkel, H. Haller, and R. Studer, *Semantic Wikipedia*, in: Web Semantics: Science, Services and Agents on the World Wide Web, vol. 5, 2007, pp. 251–261.

[13] Eclipse Modeling Framework (EMF), http://www.eclipse.org/modeling/emf/.

[14] Adapter pattern, http://en.wikipedia.org/wiki/Adapter_pattern.

[15] M. Grove, *Empire: RDF & SPARQL Meet JPA*, Semantic Universe, http://www.semanticuniverse.com/articles-empire-where-rdf-sparql-meet-java-persistence-api.html, 2010.

[16] R. Biswas and E. Ort, *The Java Persistence API - A Simpler Programming Model for Entity Persistence*, http://java.sun.com/developer/technicalArticles/J2EE/jpa/, 2006.

[17] EMFTriple, http://code.google.com/p/emftriple/.

[18] Callimachus, http://callimachusproject.org.

[19] Infinispan, http://jboss.org/infinispan/.

[20] T. Reenskaug, *MVC XEROX PARC 1978-79*, http://heim.ifi.uio.no/ trygver/themes/mvc/mvc-index.html.

[21] Bigdata, http://www.bigdata.com/.

[22] Eclipse Public License (EPL), http://www.opensource.org/licenses/eclipse-1.0.php.

[23] T. Reenskaug and J. O. Coplien, *The DCI Architecture: A New Vision of Object-Oriented Programming*, http://www.artima.com/articles/dci_vision.html.