Mooop – A Hybrid Integration of OWL and Java

Christoph Frenzel[1], Bijan Parsia[2], Ulrike Sattler[2], and Bernhard Bauer[1]

[1] Institute of Computer Science, University of Augsburg, Germany
{christoph.frenzel,bernhard.bauer}@informatik.uni-augsburg.de
[2] School of Computer Science, The University of Manchester, UK
{bparsia,sattler}@cs.man.ac.uk

Abstract. Java is a widespread object-oriented programming language for implementing information systems because it provides means to express various domains of interest. Nevertheless, some fields like Health Care and Life Sciences are so complex that Java is not suited for their design. In comparison, the Web Ontology Language (OWL) provides various powerful modelling constructs and is used to formulate large, well-established ontologies of these domains. OWL cannot, however, be used alone to build applications. Therefore, an integration of both languages, which leverages the advantages of each, is desirable, yet not easy to accomplish. We present Mooop (Merging OWL and Object-Oriented Programming), an approach for the hybrid integration of OWL ontologies into Java systems. It introduces hybrid objects, which represent both an OWL and Java entity. We have developed a prototype of Mooop and evaluated it in a case study.

Keywords: Hybrid integration, Ontologies, OWL, Object-oriented programming, Java, Ontology-based applications.

1 Motivation

Object-oriented programming languages (OOPLs) like Java are a widely adopted technology for implementing ever bigger and more complex object-oriented (OO) information systems. They provide means for the expression of the structure, i.e., possible entities and their features, and the behaviour, i.e., possible modifications of the entities at run-time, of complex problem domains. Nevertheless, some problem fields, e.g., Health Care and Life Sciences, are so complex that OOPLs are not suited to design them. For instance, anatomical models are often characterised by a vast number of concepts with complex constraints and are laborious to express in Java. In comparison, the Web Ontology Language (OWL) provides expressive modelling constructs which have been used by domain experts to formulate large ontologies of these domains, e.g., GALEN[1] and SNOMED CT[2]. Additionally, OWL allows reasoning, i.e., the inference of implicit from explicit knowledge. However, OWL does not allow to implement dynamic behaviour and, thus, it alone cannot be used to build applications.

[1] http://www.opengalen.org/index.html
[2] http://www.ihtsdo.org/snomed-ct/

C. Salinesi and O. Pastor (Eds.): CAiSE 2011 Workshops, LNBIP 83, pp. 437–447, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Ontology-based applications leverage the advantages of both paradigms by combining them. They are characterised by a complex OWL ontology and a sophisticated OOPL-expressed behaviour which regularly conducts reasoning in order to derive new knowledge form the ontology. Examples are the Patient Chronicle Model [13], and the context-aware application framework [15]. In comparison to typical semantic web applications like the FOAFMap [12], ontology-based applications require a tight integration of OWL and the OOPL. However, such integration is not easy to accomplish because differences in their semantics induce an impedance mismatch that needs to be taken into account [9].

This paper presents our novel approach for the integration of OWL ontologies into OO systems: Mooop (Merging OWL and Object-Oriented Programming). We introduce hybrid objects which are entities present in both the OWL ontology and the OO system. Additionally, we provide a flexible mechanism for linking both paradigms. In this way, Mooop creates a coherent hybrid model. This paper is based on [3] which provides more detailed information on Mooop.

Although the Mooop concept is not language specific, in this paper, we concentrate on the mainstream OOPL Java in order to exemplify the concepts. Furthermore, we use OWL 2 and the syntax introduced in [5].

2 Java and OWL

Java is a strongly typed, class-based OOPL [4]. Its basic elements are objects which exchange messages. Objects have a fixed and unique object identifier (OID), a structure defined by a collection of attributes and corresponding values, and a behaviour defined by a collection of methods. An object is created by instantiating a class which determines the structure and behaviour of the object. Java allows single inheritance, i.e., a class can inherit the structure and behaviour of at most one other class and, thus, can become a subclass. This allows the subclass to extend or overwrite the superclass. The type of an object is defined at compile-time and cannot change at run-time. Furthermore, it is forbidden to call an undefined method of an object. The methods are implemented in an imperative, Turing complete language. Java offers dynamic binding based on single dispatch which enables polymorphism.

OWL [16] is a modelling language based on Description Logic (DL) [1] for capturing knowledge in an ontology. It distinguishes between a concept level (TBox) and an instance level (ABox). The former defines OWL classes and OWL properties, and the latter OWL individuals. OWL allows the definition of atomic and complex classes. The latter are combinations of atomic classes and property restrictions. Both class types can be used in a subclass or equivalent class definition. A property is a first class citizen of an ontology and can be a subproperty or equivalent property of another property. OWL distinguishes between object properties and data properties. An OWL individual can have several explicit and implicit types, i.e., atomic or complex classes. In summary, OWL is very expressive concerning the structural features of a domain. In contrast to that, OWL cannot express any behaviour at all, and, thus, cannot be used alone to implement applications.

An important feature of OWL is reasoning, i.e., the inference of implicit knowledge from explicit knowledge, through reasoning services, e.g., consistency checking and classification. This enables post-coordination [6, p. 91]: not all concepts of the real world are modelled in the ontology as atomic classes, but the user of the system defines the concepts at run-time as anonymous complex classes. For instance, an OWL individual can have the types $\mathsf{Allergy}$ and $\exists\mathsf{causedBy.Nuts}$ simultaneously, thus, defining a not explicitly modelled nut allergy. In order to use post-coordination, the possibilities for defining complex classes at run-time should be restricted in a domain specific manner: it should only be possible to create reasonable concepts. This is called sanctioning [2].

3 Existing Integration Approaches

There are numerous existing approaches for integrating OWL into Java or similar OOPLs. Based on [13], we distinguish direct, indirect, and hybrid integration.

The direct integration represents OWL classes as Java classes and OWL individuals as Java objects. Hence, this approach offers the developers a domain-specific application programming interface (API) and type-safety which eases the application development. However, this comes at the price of limited reasoning capabilities: since Java classes cannot dynamically change their attributes at run-time, it is not possible to represent new, inferred OWL properties at run-time. Representatives of this category are So(m)mer [14] and Jastor [7].

The indirect integration utilises Java classes to represent the metaclasses of OWL, e.g., `OWLClass` or `OWLIndividual`. The OWL ontology is represented by instances of these Java classes, i.e., an object can represent, e.g., an OWL class, OWL property, or OWL individual. On the one hand, this provides run-time flexibility allowing a sophisticated reasoning. For instance, if a new OWL property instance is inferred then a new instance of the metaclass `OWLPropertyInstance` has to be created and linked with other objects. On the other hand, the development of complex software is complicated because of the generic, domain-neutral API. This approach is used by OWL API [10] and Jena [8].

The idea of the hybrid integration is to integrate a few top level concepts, i.e., very important core concepts, directly, and the vast number of specialised concepts indirectly [13]. Hence, a Java class can represent either an OWL metaclass or an OWL class, and a Java object can represent either an OWL class, an OWL property, or an OWL individual. In this way, it combines a domain-specific API and type-safety with great run-time flexibility. The disadvantage of the approach is a complexity overhead [13]. Representatives of this category are the Core Model-Builder [13] and TwoUse [11].

In order to exemplify the hybrid integration, Fig. 1 depicts a possible Java class model for a hybrid integration of the following ontology:

$$\begin{aligned} \mathsf{Pizza} &\sqsubseteq \exists\mathsf{hasName.String} \sqcap \exists\mathsf{hasTopping.Topping} \\ \mathsf{Topping} &\sqsubseteq \exists\mathsf{hasName.String} \\ \mathsf{PricedPizza} &\sqsubseteq \mathsf{Pizza} \sqcap \exists\mathsf{hasPrice.Integer}\ . \end{aligned}$$

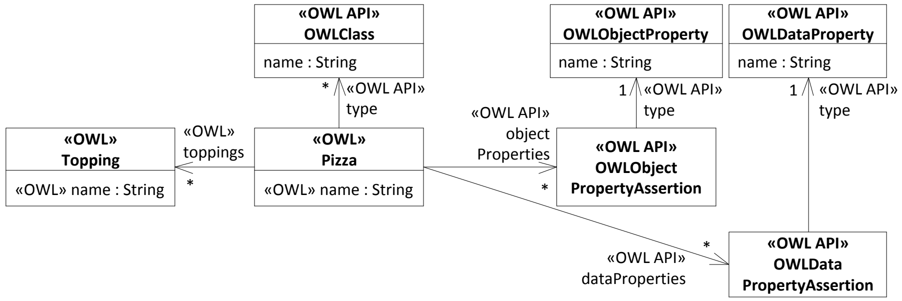

Fig. 1. Java class model for the hybrid integration of the example ontology

All directly integrated concepts are stereotyped with *OWL* and metaclasses for the indirect integration are stereotyped with *OWL API*.[3] `Pizza` and `Topping` are directly integrated top level OWL classes and have the directly integrated property `name`. `Pizza` has also indirect properties and types. The OWL class PricedPizza is integrated indirectly and, hence, is represented as an instance of `OWLClass` at run-time.

Hybrid integration is the most powerful integration approach [13]. Because of its numerous features, it is especially suited for the development of ontology-based applications. However, in order to match a wide range of requirements, the integration semantics has to be adaptable to the needs of a specific application. Integration semantics refers to the interpretation of the OWL model within Java. For example, it should be possible to implement a specific sanctioning mechanism which determines the structure of a Java object representing an OWL entity. Assume an OWL class Pizza with Pizza $\sqsubseteq$ ∃hasPrice.integer, then the sanctioning mechanism can define that a Java class representing the Pizza has a single-valued attribute `price` of type `Integer`. However, OWL would also allow Pizza individuals without an asserted price or with several prices. Obviously, such mechanisms are domain specific and difficult to generalise.

Current hybrid integration frameworks are limited in their applicability. TwoUse is actually a model-driven development framework allowing the design of a hybrid model. It allows only OWL classes to be integrated indirectly. Therefore, it is not possible to infer new properties of an OWL individual at run-time. The Core Model-Builder is, to the best of our knowledge, currently the most sophisticated framework for the hybrid integration of an OWL ontology into Java and offers a lot of features like hiding of OWL concepts, dynamic constraints on indirectly integrated attributes, and complex sanctioning. However, the adaptation of the integration is very complex since the integration semantics is spread all over the framework and, hence, numerous classes have to be changed.

[3] This convention will be used throughout this paper.

4 The Mooop Integration Approach

The Mooop integration approach was developed with the aim to create a powerful hybrid integration framework targeting ontology-based applications. Thereby, we focused on the ability to adapt the integration semantics to the requirements of a particular application. In this way, Mooop overcomes the shortcomings of current hybrid integration approaches. In order to accomplish these goals, Mooop splits the integration into the three layers shown in Fig. 2:

The OwlFrame indirectly integrates an OWL individual into Java for the domain neutral representation of knowledge from an OWL ontology.

The Mapping is an adaptable link between the OWL ontology and the OwlFrame and determines the integration semantics.

The Binding defines the hybrid integration of the OwlFrame into the target Java application. It enables the definition of hybrid classes, which represent OWL classes in Java, through Java annotations. Their instances, called hybrid objects, are present in both the OWL ontology and the Java model.

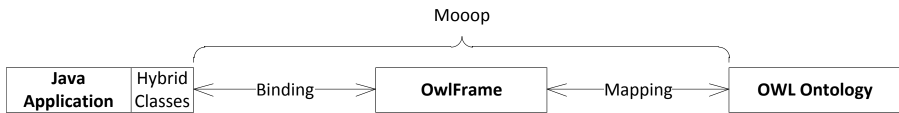

Fig. 2. The three conceptual layers of Mooop: OwlFrame, mapping, and binding

4.1 OwlFrame

Figure 3 depicts the OwlFrame as an indirectly integrated OWL individual. Notice that the association between OwlFrames and OWL individuals is injective. Since an OWL individual can have multiple types, an OwlFrame can also be associated to several type `OWLClass` objects, which indirectly integrate atomic and complex OWL classes. This enables complex post-coordination because it allows multiple typing using complex OWL classes. There are two different types of properties of an OwlFrame: `OwlObjectPropertyInstance` objects represent values for an OWL object property, i.e., OWL individuals, and `OwlDataPropertyInstance` objects represent values for an OWL data property, i.e., OWL literals.

Mooop distinguishes between asserted types and properties which express explicit knowledge about an OWL individual, and inferred types and properties which represent implicit knowledge. Thereby, the inferred information is read-only. The bound types of an OwlFrame are utilised by the mapping and binding for offering type safety (see Sect. 4.2). The method `classify()` triggers the classification of the assigned OWL individual by a reasoner. The classification result, i.e., inferred knowledge, will usually be represented as inferred types and properties.

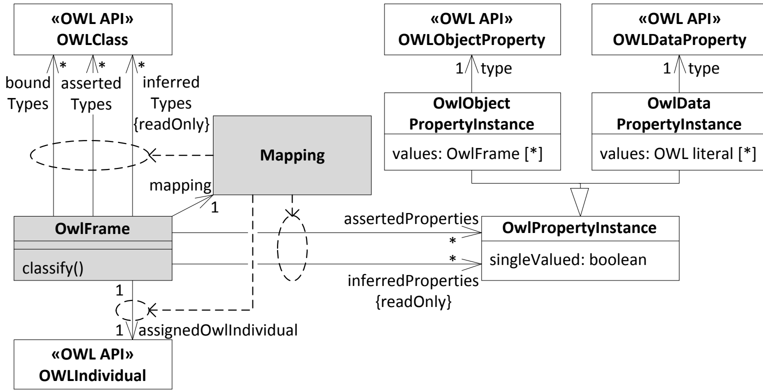

Fig. 3. The structure of the OwlFrame

In order to exemplify the OwlFrame's structure, Fig. 4 depicts a representation of an OWL individual pizza. Notice that this is just one possibility since the actual property values and types of an OwlFrame depend on the mapping. The pizza is defined by the following ontology:

$$\begin{aligned}
\mathsf{pizza} &\in \mathsf{Pizza} \\
\mathsf{mozzarella} &\in \mathsf{Mozzarella} \\
\mathsf{tomato} &\in \mathsf{Tomato} \\
\langle \mathsf{pizza}, \mathsf{mozzarella} \rangle &\in \mathsf{hasTopping} \\
\langle \mathsf{pizza}, \mathsf{tomato} \rangle &\in \mathsf{hasTopping} \\
\mathsf{PizzaMargherita} &= \mathsf{Pizza} \sqcap \exists \mathsf{hasTopping.Mozzarella} \sqcap \exists \mathsf{hasTopping.Tomato}\ .
\end{aligned}$$

4.2 Mapping

The mapping links the OWL ontology and the OwlFrame, thereby determining the integration semantics. As shown in Fig. 3, an OwlFrame is associated with exactly one mapping which defines all structural features of an OwlFrame as well as their manipulations, e.g., the addition and deletion of types or property values. Furthermore, the mapping controls the reasoning process of an OwlFrame, and the life cycle of the assigned OWL individual, i.e., the creation, loading, and deletion. From a technical point of view, the mapping is the implementation of a specific interface, whose methods are called by the OwlFrame at run-time in order to access the ontology. The implementation, however, can be customised and, thus, adapted to the application's requirements.

The bound types can be seen as a set of special asserted types which offer type safety for hybrid objects. Thus, a special kind of type safety, strong type

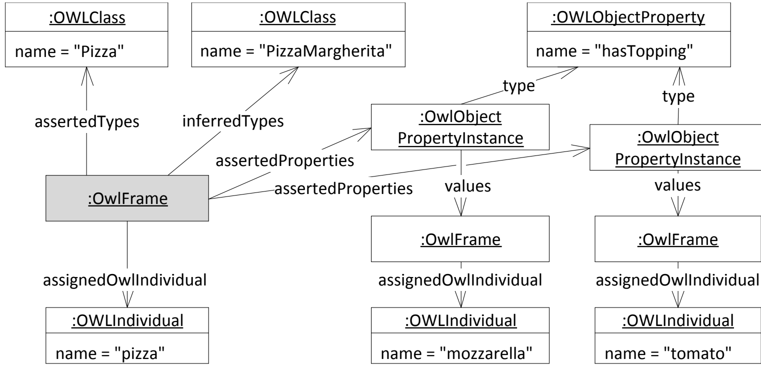

Fig. 4. The object model of an OwlFrame representing the OWL individual pizza

safety, can ensure that the asserted types of an OwlFrame are only subclasses or superclasses of the bound types. For instance, an OwlFrame `o` with the bound type Pizza cannot be asserted to be of type Vehicle if Vehicle is neither a subclass nor superclass of Pizza. In this way, it is guaranteed that an OwlFrame is specialised by its types in a reasonable manner. However, this logic can easily be changed as well.

The properties and property values of an OwlFrame are very likely to be customised, e.g., for implementing a specific sanctioning mechanism. For instance, assume the following OWL ontology:

$$\begin{aligned} \mathsf{Pizza} &\sqsubseteq \exists \mathsf{hasTopping}.\top \sqcap \exists \mathsf{hasBase}.\top \\ \mathsf{pizza} &\in \mathsf{Pizza} \\ \langle \mathsf{pizza}, \mathsf{mozzarella} \rangle &\in \mathsf{hasTopping} \ . \end{aligned}$$

The mapping can define that an OwlFrame representing pizza has an asserted property of type hasTopping with the value mozzarella, and an inferred property of type hasBase with the value `null`. Hence, the asserted and inferred properties together strictly conform to the definition of the OWL class Pizza.

4.3 Binding

The binding allows the definition of a hybrid integration by binding a Java class `C` to a specific OWL class O and attributes of `C` to information contained in an OwlFrame, e.g., the property values or the types of the OwlFrame. For instance, a Java class `Pizza` can be bound to the OWL class Pizza and the Java attribute `price` can be bound to the values of the OWL property hasPrice. The class `C` is a hybrid class and its instances are hybrid objects. Thereby, hybrid refers to their presence in both the OWL ontology and Java. Figure 5 depicts the relation between hybrid classes, which are stereotyped with *HybridClass*, and

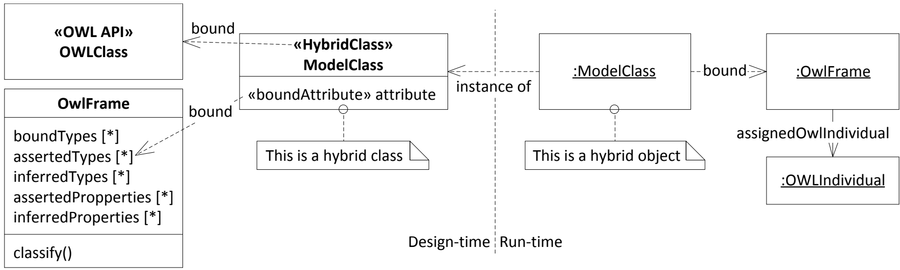

Fig. 5. A hybrid class is bound to an OWL class, its attributes are bound to features of the OwlFrame, and a hybrid object is bound to an OwlFrame object

hybrid objects. One can think of a hybrid class as the definition of a domain specific view on the information contained in an OwlFrame.

Upon instantiation of a hybrid class `C` which is bound to the OWL class O, the resulting hybrid object `c` is bound to an OwlFrame `o`. Thereby, O is added to the bound types of `o`. Furthermore, if `C` is a subclass of `D` which is bound to the OWL class P, then P is also added to the bound types of `o`. At run-time, several hybrid objects can be bound to the same OwlFrame object. For instance, assume an OwlFrame `o` that has the OWL classes O_1 and O_2 as types, and the Java classes C_1 and C_2 which are bound to O_1 and O_2, respectively. Then it is possible that a hybrid object of the Java class C_1 is bound to `o` while, at the same time, a hybrid object of the Java class C_2 is bound to `o`. As a result, both hybrid objects bound to the same OwlFrame share the same ontological information.

Mooop provides an OWL ontology-based dynamic method dispatch by annotating a method of a hybrid class to be overwritten by another method of that class if the bound OwlFrame has a specific type. For instance, assume the hybrid class `Offer` bound to the OWL class Offer. `Offer` has the method `getPrice()` which returns the value for the OWL property hasPrice. However, a SpecialOffer with SpecialOffer $\sqsubseteq$ Offer has an additional discount. Hence, `Offer` has a method `getDiscountedPrice()` which calculates a discounted price. Now, `getPrice()` is annotated to be overwritten by `getDiscountedPrice()` if the hybrid object has the type SpecialOffer.

The implementation of the binding in the Mooop framework is inspired by the Java Persistence API (JPA)[4]. It utilises Java annotations to allow a declarative definition of hybrid classes. The listing in Sec. 5 exemplifies this.

5 Case Study

We have developed a case study to show the feasibility of the Mooop approach: the pizza configurator which is inspired by The Manchester Pizza Finder[5]

[4] http://jcp.org/aboutJava/communityprocess/final/jsr317/index.html
[5] http://www.co-ode.org/downloads/pizzafinder

developed by Matthew Horridge. The pizza configurator offers the possibility to create a pizza at one's own option by adding desired pizza toppings to an initially empty pizza base. Thereby, the system has to ensure that there is at most one topping of each kind on the pizza. When the user has finished the customisation, the pizza is classified and its price is calculated: this can either be the sum of the prices of the selected toppings or, if it matches a predefined type of pizza, the price of that pizza type. For example, assume a pizza base is topped with mozzarella and tomatoes, then the system calculates the price of the pizza as the price of a pizza Margherita. Furthermore, the user can be presented additional information about the pizza like its spiciness. While the structure of this example has already been defined in the famous Pizza Ontology[6], the behaviour cannot be expressed with the means of OWL but has to be implemented in Java. Therefore, Mooop will be used to create a coherent hybrid model integrating both OWL and Java.

As with most hybrid integration approaches, the developer would start the implementation of the pizza configurator by picking several top level classes from the OWL ontology and creating hybrid classes for them. In our case the hybrid class `Pizza` is bound to the OWL class Pizza, and the hybrid class `Topping` to Topping. Afterwards the programmer defines the structure of the classes using bound attributes: the Java class `Pizza` gets the attributes `toppings` which is bound to the OWL property hasTopping, and `price` which is bound to hasPrice. Furthermore, an attribute `properties` is created in `Pizza` which contains additional properties of the bound OwlFrame, e.g., the origin of the pizza, and, thus, allows an indirect access to them.

The following listing shows an extract from the `Pizza` class. The method `calculatePrice()` sums up the prices of the toppings of a pizza. However, using the dynamic OWL ontology-based method dispatch, this method is overwritten by the method `getPrice()` if the pizza is a PricedPizza. Notice that the implementation of the annotated methods is ignored since the calls of these methods are intercepted and processed by the binding of Mooop.

```
@HybridClass("Pizza")
public class Pizza {
  @OwlProperty("hasTopping")
  public void addTopping(final Topping topping) { }
  @OwlProperty(value="hasTopping", valueType=Topping.class)
  public Set<Topping> getToppings() {
    return null;
  }
  @OwlDispatch({ @OwlDispatchEntry(owlType="PricedPizza",
    methodName="getPrice") })
  public Integer calculatePrice() {
    /* sum up prices ... */
  }
```

[6] `http://www.co-ode.org/ontologies/pizza/pizza-latest.owl`

```
  @Classify
  public void classify( ) { }
  @OwlProperty(value="hasPrice", valueType=Integer.class)
  protected Integer getPrice() {
    return null;
  }
  @OwlProperty()
  public Map<String, Set<Object>> getProperties() {
    return null;
  }
  ...
}
```

A custom mapping is employed to dynamically create a covering axiom for a pizza: assume the OwlFrame o with the type Pizza and the toppings Tomato and Mozzarella. In this case, the custom mapping adds an axiom stating that the assigned OWL individual of o has only the toppings Tomato and Mozzarella. This is necessary to enable the reasoner to infer the right type of pizza, here PizzaMargherita. The custom mapping is implemented by simply extending a default mapping class and overwriting the methods which define the properties.

The pizza configurator case study also shows the advantages of Mooop compared to other hybrid integration frameworks: TwoUse allows only a limited and inflexible integration, since only OWL classes can be integrated indirectly. Hence, it is not possible to define indirectly integrated OWL properties like the `properties` attribute in the class `Pizza`. Furthermore, it offers no means to change the integration semantics and, thus, the logic of the custom mapping of the pizza configurator would have to be mixed up with business logic. The integration features offered by the Core Model-Builder and Mooop are comparably. Accordingly, the Core Model-Builder allows defining indirectly integrated properties. However, the adaptation of the integration semantics is much more complicated since the logic is spread over a lot of classes. Thus, the custom mapping of the pizza configurator would have called for a complex adaptation of several classes in the Core Model-Builder framework.

6 Conclusion and Outlook

This paper presents Mooop, an approach for the hybrid integration of OWL into Java. Thereby, Mooop exploits the strengths of direct and indirect integration and, thus, is both powerful and easy to use. In contrast to other hybrid integration approaches, Mooop allows an easy customisation of the integration by introducing three layers: the OwlFrame is an indirect Java representation of ontological knowledge, the mapping is a customisable link between an OWL ontology and the OwlFrame, and the binding defines the hybrid integration of the OwlFrame into the target Java application through hybrid classes. Because of its flexibility, we think that Mooop can facilitate the broad application of OWL as a knowledge representation language for information systems. We have shown the advantages of Mooop in a case study.

In the future, the development of more sophisticated mappings and bindings is probably the most pressing issue. For instance, mappings for standard OWL modelling guidelines can extend the reuse immensely. Another subject which has not been investigated yet is the storage of the run-time model of a Mooop system which consists of OWL individuals and Java objects. Furthermore, a methodology for modelling hybrid models with Mooop is necessary.

The basic idea of Mooop, i.e., the division of the hybrid integration of OWL and Java into OwlFrame, mapping, and binding, can be generalised into a concept for a generic hybrid integration of any modelling approach, e.g., Topic Maps[7], into Java. It is an interesting future research topic to design, implement, and evaluate such a general concept.

References

1. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press, New York (2003)
2. Bechhofer, S., Goble, C.A.: Using a Description Logic to Drive Query Interfaces. In: International Workshop on Description Logics (1997)
3. Frenzel, C.: Mooop – A Generic Integration of Object-Oriented and Ontological Models. Master's thesis, University of Augsburg (2010)
4. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, 3rd edn. Addison-Wesley, Boston (2005)
5. Horrocks, I., Patel-Schneider, P.F., van Harmelen, F.: From $\mathcal{SHIQ}$ and RDF to OWL: The Making of a Web Ontology Language. J. Web Sem. 1, 7–26 (2003)
6. International Health Terminology Standards Development Organisation: SNOMED Clinical Terms User Guide. Technical report (2010)
7. Jastor home page, `http://jastor.sourceforge.net/`
8. Jena Semantic Web Framework home page, `http://jena.sourceforge.net/`
9. Oren, E., Delbru, R., Gerke, S., Haller, A., Decker, S.: ActiveRDF: Object-Oriented Semantic Web Programming. In: 16th International Conference on World Wide Web, pp. 817–824. ACM, New York (2007)
10. OWL API home page, `http://owlapi.sourceforge.net/`
11. Parreiras, F.S., Staab, S.: Using Ontologies with UML Class-based Modeling: The TwoUse Approach. J. Data Knowl. Eng. 69, 1194–1207 (2010)
12. Passant, A.: FOAFMap: Web 2.0 meets the Semantic Web. In: 2nd Workshop on Scripting for the Semantic Web, pp. 67–68 (2006)
13. Puleston, C., Parsia, B., Cunningham, J., Rector, A.L.: Integrating Object-Oriented and Ontological Representations: A Case Study in Java and OWL. In: Sheth, A.P., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T.W., Thirunarayan, K. (eds.) ISWC 2008. LNCS, vol. 5318, pp. 130–145. Springer, Heidelberg (2008)
14. So(m)mer home page, `http://java.net/projects/sommer`
15. Springer, T., Turhan, A.Y.: Employing Description Logics in Ambient Intelligence for Modeling and Reasoning about Complex Situations. J. Ambient Intell. Smart Environ. 1, 235–259 (2009)
16. W3C OWL Working Group: OWL 2 Web Ontology Language Document Overview. W3C Recommendation (October 27, 2009)

[7] `http://www.isotopicmaps.org/`