A Metamodel Approach to Context Information

Florian Fuchs, Iris Hochstatter, Michael Krause
Mobile and Distributed Systems Group, Institute for Informatics
Ludwig Maximilian University Munich
Oettingenstr. 67, 80538 Munich, Germany
{florian.fuchs|iris.hochstatter|michael.krause}@nm.ifi.lmu.de

Michael Berger
Siemens AG Corporate Technology
Otto-Hahn-Ring 6
81739 Munich, Germany
m.berger@siemens.com

Abstract

In order to enable a common understanding of context information we introduce a modeling concept that embraces four abstraction layers from meta-metamodel and metamodel to model and instance layer. Being compliant to OWL DL we especially consider the ability of reasoning over the information and the modeling of quality attributes.

1 Introduction

Context-aware services (CASs) nowadays are often directly linked with their context sources, they operate in closed systems. With the advance of context-aware computing, the need to unify context representation arises. Formal context models will facilitate the exchange of context information, provide a common understanding and thus enable interoperability between different context-aware services and context sources.

Whilst it is agreed upon that the use of ontologies is a feasible approach to share the understanding of context information (cp. [12]) and miscellaneous collections of ontologies arise, we propose an even more basic proceeding for context modeling with ontologies. We are developing a formal metamodel that specifies the construction of restricted OWL DL-based ontologies. OWL DL is one of the three sublanguages of the *Web Ontology Language (OWL)*, a W3C recommendation [11]. It is an implementation of Description Logics as a trade-off between maximum expressiveness, computational complexity and decidability. The information modeling we propose takes into account the requirements posed on a context model by the special characteristics of context information and context-aware systems, e.g. it incorporates the specification of quality considerations of context and association rules. Tools supporting the development of context-aware services can then be designed based on our formal context modeling approach. Such a tool can generate an ontology in OWL DL from a graphical representation generated by the developer.

Section 2 describes our approach to metamodeling context starting from the requirements it has to fulfill. Our implementation of a sample application is shown in section 3 and our experiences with it are discussed. We conclude the paper with a short summary and an outlook to future work.

2 Context Metamodeling Approach

Recent approaches to context modeling used very different data structures as their basis, a detailed discussion can be found in [12]. However, the research community seems to agree upon that ontologies are a very promising formalism for modeling context (cp. [12] and [9]). In [12], Strang identified six important requirements for a context model in a ubiquitous computing environment: distributed compostition, partial validation, richness and quality of information, incompleteness and ambiguity, level of formality, and applicability to existing environments. Ontologies fulfills all those requirements in contrast to the other solutions that mostly lack in support for richness and quality of information as well as assistance with incompleteness and ambiguity of information. Our modeling approach was driven by requirements we collected during the work with and development of a reference context brokering and processing architecture (see Context Composition infrastructure in [4]). We identified the following additional requirements:

1. *Evolutionary development:* The individual developer has the best knowledge about the service he designs and implements. Context models for different services may often differ and can never be exhaustive and completed. Therefore, a modeling technique should support adaptability and evolutionary design. At the same time, there will be (and there partly already are) standard models for the most important information and services used. It has to be possible to reuse and interoperate with such standard models.

2. *Interoperability:* There are already models of the world and special context models around. Thus, it is important to consider interoperability with existing models or keep mappings in mind when designing a modeling technique for context information.

3. *Reasoning:* Context information is gathered from many different sources. The interesting context information often cannot be sensed directly. Composition of context information, as Strang says, should be possible in a distributed way. In addition, deriving new information from available context information should be feasible and has to be considered in the modeling technique.

4. *Inferential efficiency:* Context models are the basis for the application logic of a context-aware system. By reasoning on the available context information, an application can adapt its behaviour accordingly. That is why a modeling technique for context information should provide modeling constructs that allow for efficient inference and reasoning operations.

5. *Ease of use:* There will be numerous heterogenous context-aware applications in the future and context-aware applications may be targeted at special small groups or even individuals. Change will be immanent and especially developers have to be supported when developing those applications. Therefore, a modeling technique should be easy to use and the models constructed with it should be concise.

Many research groups developed particular ontologies for context and context-aware services (e.g. Chen et al. [6], Wang et al. [8]). To compute all information complying to different ontologies, there has to be a common structure underlying. Thus, we pursue a more formal approach to support reasoning and follow a widely-used metamodeling proceeding that embraces four layers. With our formal approach on the one hand we restrict the interoperability of services to those that use information models that comply with our metamodel directly or where a respective mapping can be found. On the other hand we ensure therewith that tools, decision and reasoning algorithms can be fully applied and thus increase the degree of interoperability.

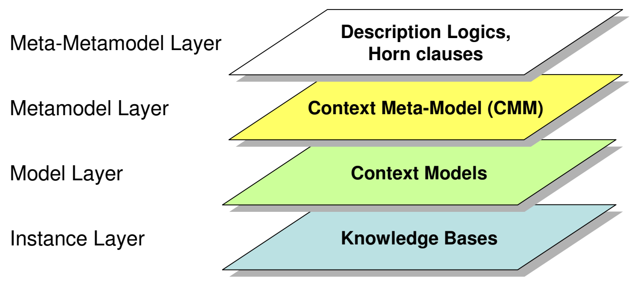

Figure 1. Conceptual architecture

2.1 Modeling Architecture

As a basis for the formal specification of our context modeling technique we adopt a widely-used conceptual metamodeling architecture, where elements in a given conceptual layer describe elements in the next layer down. It comprises the following four layers (see also figure 1):

1. *Meta-metamodel layer*: Provides the formal constructs for specifying our Context MetaModel (CMM).

2. *Metamodel layer*: Provides the modeling constructs of our Context MetaModel.

3. *Model layer*: Uses our Context MetaModel to build custom ontologies.

4. *Instance layer*: Implements a custom context model in a knowledge base (KB).

The following sections follow this metamodeling architecture in order to provide a structured and formally clean specification of our modeling constructs.

2.2 Choosing the Meta-Metamodel Layer

On the meta-metamodel layer we select Description Logics for our terminology and Horn clauses (a special class of formulae which are of particular interest for logic of programming) for rules as a formal foundation. Description Logics provide the formal semantics for specifying ontologies of terms and their relations [2]. Their main benefit is that they balance expressiveness and inferential efficiency, which is a basic requirement for effective context modeling.

2.3 Modeling Constructs on the Metamodel Layer

For the formal specification of our context metamodel, we have leveraged the existing ontology language OWL DL

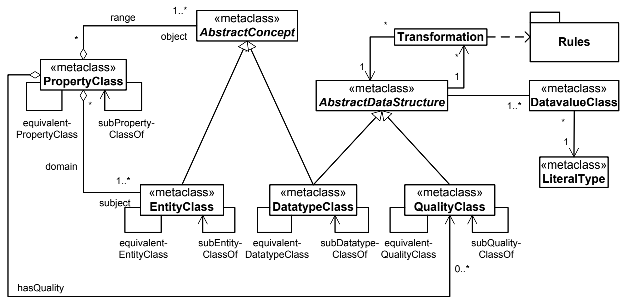

Figure 2. Metamodel layer: Illustration of the metamodeling constructs

as an implementation of the Description Logics SHION(D). Rules are formally based on first-order logic and are currently specified according to the Semantic Web Rule Language (SWRL) draft [10], a W3C member submission originating from RuleML and integrating well with OWL. We do not use OWL itself, but we have done a formal mapping from our metamodel to OWL DL.

Figure 2 shows an overview of our context modeling metamodel (note that AbstractConcept and AbstractDataStructure are declared abstract and only introduced for modeling Property associations and complex data types).

Base constructs for representing (context) knowledge are entity classes, datatype classes and properties with their associated quality classes:

- *Entity class*: base construct for representing a group of entities (persons, places, things, events etc.) that belong together because they share some properties

- *Datatype class*: base construct for representing a datatype (temperature, noise level, position etc.)

- *Property*: base construct for representing a type of relationship between an instance of an entity class and an instance of either an entity or a dataype class. An example for a *property* as a relation between two entities on the model layer is: `Person` ”owns” `MobilePhone`. `Person` ”hasPhoneNumber” `PhoneNumber` relates an entity with a datatype (more details are given in the next subsection).

  Each property has a specified *domain* and *range*, i. e. a collection of entity classes and a collection of either entity or datatype classes that specify the valid classes for the first and second instance, respectively. Each property is also associated with a collection of quality classes (see below) that specify the quality aspects that are relevant to the property.

  Additional restrictions as known from Description Logics, such as cardinality restrictions etc., may also be used for properties.

- *Quality class*: base construct for representing specific quality aspects of dynamically acquired information (certainty, precision, resolution etc.) also known as *Quality of Context* [5].

In order to represent *temporal history* information, for every property the acquisition time is captured as a timestamp. It is a mandatory quality class for every property.

Dependencies between properties are expressed as rules in the form of Horn clauses. Each rule expresses an implication between an antecedent and consequent: whenever the conditions specified in the antecedent hold, the conditions specified in the successor must also hold. This allows to specify consistency conditions as well as derivation rules. Conditions can reference entity classes and datatype classes as well as properties and their associated quality classes. This way a rule can take into account quality information and also specify the quality of the deduced properties.

In addition, there are two special constructs for the semantically rich specification of datatypes: datavalue properties and transformation rules:

- *Datavalue Class*: base construct for specifying data structures, i.e. datatype classes and quality classes. Each datavalue class associates a data structure (*AbstractDataStructure* in 2) with a literal type and thus allows to compose complex data structures from literals. E.g. the coordinates for a position are composed from `longitude` and `latitude`.

- *Transformation*: base construct for representing a transformation from values of one data structure to values of another data structure. It is a directed association between two data structures and references a rule that specifies the transformation. An example is the transformation between a position in Gauss-Krueger coordinates into a WGS-84 format, the transformation function itself is given or described in a rule in the Rules and the identifier for the rule is given in the model itself.

  Note that - in contrast to all other constructs introduced so far - the transformation is not declared as a metaclass. This is because transformations are special in that they work on classes (datatype and quality classes) instead of instances. Transformations are therefore instantiated on the model layer and not on the instance layer. This is why they represent classes and not metaclasses on the metamodel layer.

Further modeling constructs are *specialization-relations* that may be specified between two entity classes (subEntityOf), two datatype classes (subDatatypeOf), two quality

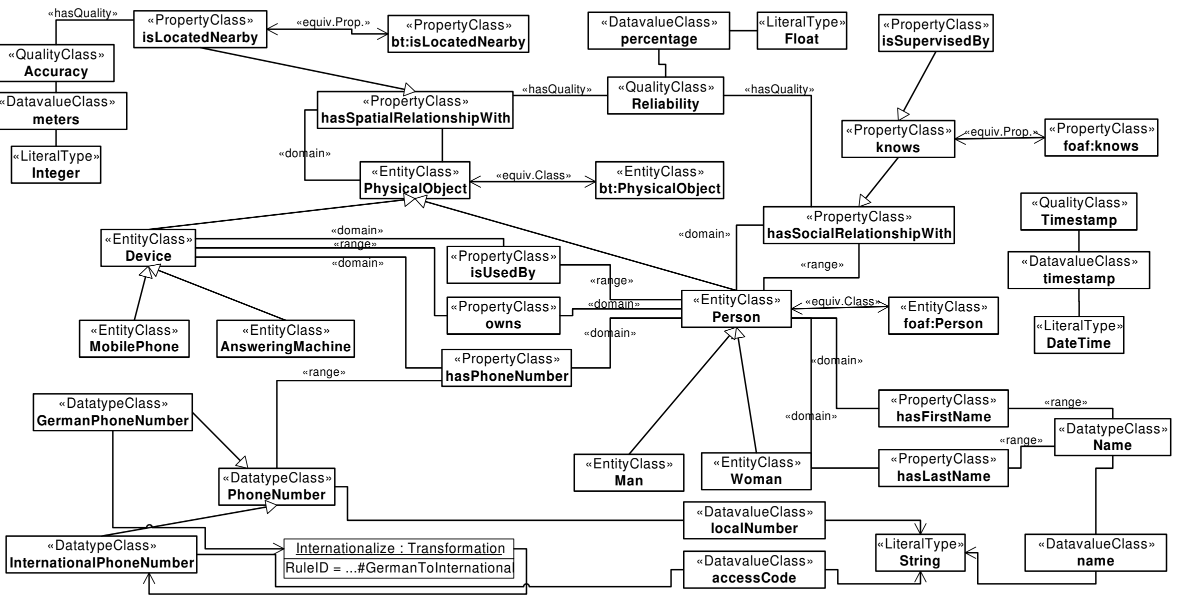
«hasQuality»
«PropertyClass» isLocatedNearby
«equiv.Prop.»
«PropertyClass» bt:isLocatedNearby
«DatavalueClass» percentage
«LiteralType» Float
«PropertyClass» isSupervisedBy
«QualityClass» Accuracy
«DatavalueClass» meters
«LiteralType» Integer
«PropertyClass» hasSpatialRelationshipWith
«hasQuality»
«QualityClass» Reliability
«hasQuality»
«PropertyClass» knows
«equiv.Prop.»
«PropertyClass» foaf:knows
«domain»
«EntityClass» PhysicalObject
«equiv.Class»
«EntityClass» bt:PhysicalObject
«QualityClass» Timestamp
«PropertyClass» hasSocialRelationshipWith
«domain»
«EntityClass» Device
«domain»
«range»
«domain»
«PropertyClass» isUsedBy
«range»
«DatavalueClass» timestamp
«range»
«PropertyClass» owns
«domain»
«EntityClass» Person
«equiv.Class»
«EntityClass» foaf:Person
«LiteralType» DateTime
«EntityClass» MobilePhone
«EntityClass» AnsweringMachine
«domain»
«PropertyClass» hasPhoneNumber
«range»
«domain»
«DatatypeClass» GermanPhoneNumber
«PropertyClass» hasFirstName
«range»
«domain»
«DatatypeClass» Name
«range»
«EntityClass» Man
«EntityClass» Woman
«PropertyClass» hasLastName
«DatatypeClass» PhoneNumber
«DatavalueClass» localNumber
«DatatypeClass» InternationalPhoneNumber
Internationalize : Transformation
RuleID = ...#GermanToInternational
«DatavalueClass» accessCode
«LiteralType» String
«DatavalueClass» name

Figure 3. Model layer: Sample context model for the intelligent answering machine application. Namespaces "per" and "foaf" stand for "http://pervasive.semanticweb.org/ont/2004/01/person" and "http://xmlns.com/foaf/0.1/"

classes (subQualityOf) or two properties (subPropertyOf) in order to organize them in (separate) specialization hierarchies. Finally, there are *equivalence-relations* that may be specified between two entity classes (equivalentEntity), two datatype classes (equivalentDatatype), two quality classes (equivalentQuality) or two properties (equivalentProperty). Their semantics is that the first node represents the equivalent concept or role as the second node and should therefore be interpreted equivalently. They are useful for mapping context models that have been developed separately in order to enable interoperability.

2.4 Building Context Models on the Model Layer

By designing a context model, the developer of a context-aware application specifies the understanding of the world that the application will have.

A developer first identifies the real-world entities that are relevant for the application in mind and models them as *entity classes*. Entity classes can be specified in terms of other entity classes and their set-theoretic operations, or they are explicitly organized in an specialization hierarchy. For example, the developer of an intelligent answering machine application may specify the entity classes like shown in figure 3: `Person` (e. g. the people calling), `Activity` (e. g. what somebody is busy with), `Device` (e. g. the machine itself) and others. Then the developer identifies the information about these entities that will be relevant for the application. That includes both information that will be acquired from the context as well as any other information. It is modeled in terms of *properties* of the relevant entity class. Properties either refer to a another *entity class* ("a `Person` knows another `Person`", "a `Person` is occupied with an `Activity`") or a *datatype class* ("a `Device` has a `PhoneNumber`"). As a specialization of the knows-property could be specified "a `Person` supervises a `Person`".

The structure of a datatype class as well as of a quality class is specified using *datavalue classes*. Abstract data structures may be linked with *transformation rules* to indicate that a transformation between the two is available. For example, the datatype class `PhoneNumber` may have only one datavalue class that specifies a string literal–the phone number represented as a string. The specialized datatype class `InternationalPhoneNumber` may have an additional datavalue class for the international access code as a string literal. For example, the quality class `Certainty` could be relevant for the property "occupied with", while both `Certainty` and `Accuracy` could be relevant for the property "located nearby".

Rules are the basis for specifying transformations between data structures, derivations of context information and consistency conditions. Transformation rules are referenced by transformations. They specify how values of one data structure can be transformed to values of another data structure. For example, a transformation rule for transforming a `GermanPhoneNumber` to `InternationalPhoneNumber` could state: If there is a `GermanPhoneNumber` with a datavalue class `localNumber`, then there is a `InternationalPhoneNumber` with the same instance of the datavalue class `localNumber` and a datavalue instance of `accessCode` with value “49”. Derivation rules describe how additional information can be generated automatically. Such a rule could for example specify that if a `Device` “has phone number” `PhoneNumber` and “is owned by” `Person` then it can be derived that this `Person` has the property “has phone number” `PhoneNumber`. The rule would also specify how the quality attributes of the resulting property are affected. Consistency rules specify conditions that must always hold for a context model or, more precisely, its instantiations. For example, a consistency rule could specify that persons who work in the same department also work for the same organization. Instead of specifying everything from scratch, context models can also partially or completely be reused. This is enabled by the mapping constructs provided by the metamodel. Two entity classes, two property classes, two datatypes classes or two quality classes from separate models can be declared representing the same using equivalence-relations. Then it is sufficient for an application to understand one of the both to also implicitly understand the other one. Having specified such a context model is the basis for the following use cases:

- *Model consistency check*: Is the specification of the context model consistent and valid with respect to the metamodel?
- *Reuse and extension*: Is there a consensus model that can be reused or that can be extended?
- *Model interoperability*: Are there consensus constructs that the new model can be mapped to in order to increase interoperability?

2.5 Using Context Models on the Instance Layer

Based on a context model, a *knowledge base* can be constructed on the instance layer, see figure 4. While the context model describes the vocabulary and the structure of the world, the knowledge base represents the actual state of the world with respect to this context model. It abstracts away from actual context sources and instead expresses context information in the terms of the context model.

Knowledge is represented as statements about entities. A statement is an instance of a property. It is always about an instance of an entity class and references either an instance of an entity class or an instance of a datatype class (according to the specification of the property). For example, the proximity of Alice to her answering machine could be expressed as: Alice, an instance of *Person*, has a property instance of “located nearby” that references Alice’s answering machine, an instance of *Device*.

Entity instances are uniquely identified in order to avoid inconsistencies and to enable knowledge base consolidation. This can be done using URIs. Every property instance is associated with the timestamp of its acquisition. For each quality class that is associated with the property class, the property instance is assigned the current value. This way freshly acquired property instances are continuously added to the knowledge base and entity instances are created as required. Based on the timestamps, a history is automatically captured. A separate *knowledge base management component* is responsible for managing the content of the knowledge base. Depending on the requirements of applications, irrelevant property instances can be cached, archived or removed. Entity instances that do not participate in any property instance can always be removed. A separate *rule engine* constantly applies the rules specified in the context model to the content of the knowledge base. This way it checks dependencies and performs derivations.

With a knowledge base like this, the following use cases are possible:

- *Information consistency check*: Is the knowledge represented in the knowledge base consistent with the specification of the underlying context model?
- *Knowledge base distribution*: With knowledge being represented as a network of (basically) property instances and entity instances, knowledge bases can be decomposed in arbitrary views. They can be deployed both locally or in the infrastructure.
- *Information interoperability*: Knowledge bases that are based on a shared context model or context models that are mapped to each other, can also share their contents. Thus, reuse of the information stored there for many purposes is possible. Consolidation of knowledge bases is facilitated by the unique identification of entity instances.
- *Reasoning*: One important use of a knowledge base will be to reason about its content. A context-aware application’s application logic is specified with respect to a context model and executed based on a corresponding knowledge base. Reasoning is facilitated by the consistent representation of heterogeneous information in accordance with a formally specified context model. By leveraging specialization hierarchies, reasoning about incomplete or vague knowledge is possible. Quality of the available knowledge can be taken into account during reasoning using the quality attributes.

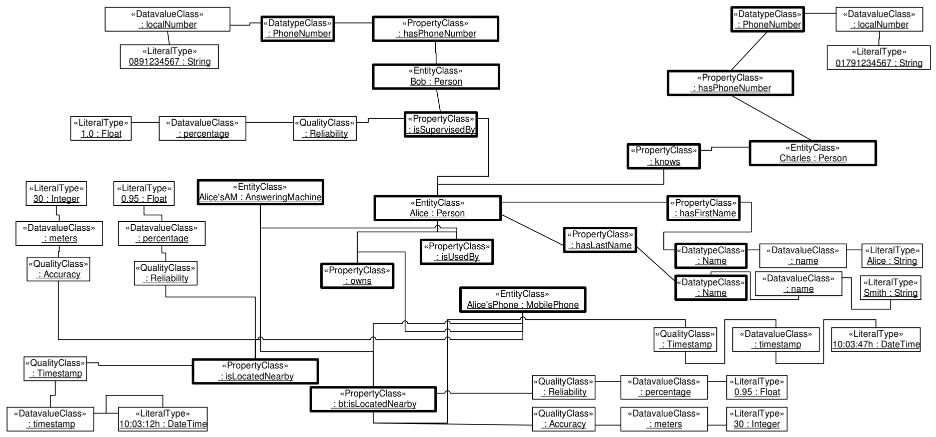
«DatavalueClass» : localNumber
«LiteralType» 0891234567 : String
«DatatypeClass» : PhoneNumber
«PropertyClass» : hasPhoneNumber
«EntityClass» Bob : Person
«LiteralType» 1.0 : Float
«DatavalueClass» : percentage
«QualityClass» : Reliability
«PropertyClass» : isSupervisedBy
«DatatypeClass» : PhoneNumber
«DatavalueClass» : localNumber
«LiteralType» 01791234567 : String
«PropertyClass» : hasPhoneNumber
«PropertyClass» : knows
«EntityClass» Charles : Person
«LiteralType» 30 : Integer
«LiteralType» 0.95 : Float
«EntityClass» Alice'sAM : AnsweringMachine
«EntityClass» Alice : Person
«PropertyClass» : hasFirstName
«DatavalueClass» : meters
«DatavalueClass» : percentage
«PropertyClass» : isUsedBy
«PropertyClass» : hasLastName
«DatatypeClass» : Name
«DatavalueClass» : name
«LiteralType» Alice : String
«QualityClass» : Accuracy
«QualityClass» : Reliability
«PropertyClass» : owns
«DatatypeClass» : Name
«DatavalueClass» : name
«LiteralType» Smith : String
«EntityClass» Alice'sPhone : MobilePhone
«QualityClass» : Timestamp
«DatavalueClass» : timestamp
«LiteralType» 10:03:47h : DateTime
«QualityClass» : Timestamp
«PropertyClass» : isLocatedNearby
«QualityClass» : Reliability
«DatavalueClass» : percentage
«LiteralType» 0.95 : Float
«PropertyClass» : bt:isLocatedNearby
«DatavalueClass» : timestamp
«LiteralType» 10:03:12h : DateTime
«QualityClass» : Accuracy
«DatavalueClass» : meters
«LiteralType» 30 : Integer

Figure 4. Instance layer: Sample snapshot of a knowledge base for the intelligent answering machine application

3 Implementation and Discussion

Based on our context modeling approach we have implemented a knowledge base engine and a sample application.

3.1 Knowledge Base Engine

The knowledge base engine handles the management of a context model and a corresponding knowledge base. Applications use it to build context models with our metamodeling constructs, have a context model instantiated in a knowledge base and query and reason about the available context information in this knowledge. This way they do not have to deal with handling context information, but can focus on designing their context model and adjusting their application logic. The knowledge base engine is implemented in Java and supports arbitrary representations of the underlying context information. The prototype implementation is based on the Web Ontology Language OWL (more precisely its sublanguage OWL DL) and the Resource Description Framework RDF (both W3C recommendations) and uses the Jena2 Semantic Web Toolkit [1] to handle OWL and RDF data.

There is no direct mapping from our metamodeling constructs to OWL DL as associating OWL classes with OWL properties (in our case quality classes with properties) is not legal in OWL DL. We therefore transform our metamodeling constructs to OWL DL constructs as follows: *Entity classes*, *datatype classes* and *quality classes* are realized as OWL classes. *Properties* and *transformation rules*, however, are also realized as OWL classes and appropriate OWL object properties connecting them to the specified entity, datatype and quality classes are automatically generated. *Datavalue classes* combined with *literal types* are realized as OWL datatype properties. *Specialization relations* are realized as OWL subClassOf-relations and *equivalence-relations* as OWL equivalentClass–relations. For a more detailed description about the representation of our Context Meta Model in OWL DL and SWRL cp. [7]. On the knowledge base layer, the OWL constructs are instantiated accordingly. Property instances (in the form of OWL individuals) can now be associate with the required acquisition timestamp and the specified quality values in compliance with OWL DL. The described transformation is also applicable to other existing OWL ontologies in order to use them for ontology mapping.

The knowledge base engine transparently handles the translation from our modeling constructs to OWL DL and vice versa. It provides interfaces for adding context information from context sources, querying the available information and refining and cleaning up information stored in the knowledge base.

3.2 Sample Application: Intelligent Answering Machine

Based on our knowledge base engine we have implemented a sample application that realizes the intelligent answering machine application. An appropriate context model specified with respect to our context metamodel is depicted in figure 3. It shows the required entity classes, datatype classes and quality classes. Properties, transformation rules and datavalue classes are displayed with their specified domain and range with respect to entity classes and datatype classes. Some of the relevant quality classes for properties are displayed, too. Note the exemplary mappings of `Person` and `knows` to the corresponding terms in the popular *Friend-of-a-Friend (FOAF)* ontology [3] and Person ontology, respectively (the latter being an OWL representation of the RDF-based FOAF ontology, provided by Chen's Standard Ontology for Ubiquitous and Pervasive Applications SOUPA [6]).

Figure 4 shows how the snapshot of a corresponding knowledge base for this context model could look like in a simplified way. The application logic of the prototype is specified with respect to the context model and executed based on the knowledge base: While the answering machine service is disabled, it monitors the knowledge base for new information about the location of its user (`hasSpatialRelationship`). When the user has left the desk (outdated `isLocatedNearby` and no other matching `hasSpatialRelationship` or one of its subproperties), it actives itself. On an incoming call it checks the knowledge base for the calling number. Depending on the deduced social relationship with the caller (query for matching `hasSocialRelationshipWith`) it plays different messages or even forwards the call to the user's mobile phone in case of an important call.

3.3 Experiences

Implementing the prototype application and modeling other case studies we have found that our context modeling approach expedites context model engineering and application development: Reusing existing consensus ontologies results in higher quality models and higher interoperability. The application logic is simplified as all relevant information is stored in the knowledge base in a consistent way and can be queried in a consistent way. The foundation on ontologies enables evolutionary development. Having the possibility of annotating every property with quality attributes allows to take into account quality in a flexible way. Including quality classes (as well as datatype classes) in context models increases the level of specification detail and facilitates interoperation. Rules prove versatile enough for expressing complex derivations, but grow complicated quickly (especially when taking into account quality). This suggests providing tool support for the specification of rules. Furthermore, realizing derivation rules is hampered by the lack of existing suitable rule engines.

Performance is sufficient for the prototype application, but could be critical with more complex applications and larger-scale knowledge bases. Maintenance and management of knowledge bases are crucial in this respect. A basis for optimizations is provided by acquisition timestamps.

4 Conclusion

We proposed a modeling technique for context information based on metamodeling which considers the requirements posed on it by the nature of context information and the development of context-aware systems. Most importantly, it keeps in mind the relation between context and its entity, takes into account quality of context, provides for interoperability, and accounts for formality.

In the near future, we plan to support developers with a tool that accounts for our modeling approach and lets them design their own ontologies, relate to exisiting ontologies and thus eases the development of context-aware services. Also, the transparent distribution and collection of context information in knowledge bases is a future goal.

References

[1] Jena: A Semantic Web Framework for Java.

[2] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[3] D. Brickley and L. Miller. FOAF Vocabulary Specification, 2004.

[4] T. Buchholz, M. Krause, C. Linnhoff-Popien, and M. Schiffers. Coco: Dynamic composition of context information. In *Proceedings of the First Annual International Conference on Mobile and Ubiquitous Computing (MobiQuitous) 2004*, Boston, Massachusetts, USA, aug 2004. IEEE.

[5] T. Buchholz, A. Küpper, and M. Schiffers. Quality of Context: What it is and why we need it. In *10th Workshop of the HP OpenView University Association (HPOVUA'03), Geneva, Switzerland*, 2003.

[6] H. Chen, F. Perich, T. Finin, and A. Joshi. SOUPA: Standard Ontology for Ubiquitous and Pervasive Applications. In *International Conference on Mobile and Ubiquitous Systems: Networking and Services*, Boston, MA, August 2004.

[7] F. Fuchs. A modeling technique for context information. Master's thesis, Ludwig Maximilian University Munich, 2004.

[8] T. Gu, H. Wang, H. K. Pung, and D. Q. Zhang. An ontology-based context model in intelligent environments. In *Proceedings of Communication Networks and Distributed Systems Modeling and Simulation Conference*, 2004.

[9] K. Henricksen, S. Livingstone, and J. Indulska. Towards a hybrid approach to context modelling, reasoning, and interoperation. In *Proceedings of the First International Workshop on Advanced Context Modelling, Reasoning And Management, UbiComp'2004*, September 2004.

[10] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. W3C Member Submission, 2004.

[11] D. L. McGuinness and F. van Harmelen. OWL Web Ontology Language Overview. W3C Recommendation, 2004.

[12] T. Strang and C. Linnhoff-Popien. A context modeling survey. In *Proceedings of the Workshop on Advanced Context Modelling, Reasoning and Management*, 2004.