Noname manuscript No.
(will be inserted by the editor)

Managing Multiple and Distributed Ontologies in the Semantic Web

A. Maedche and B. Motik and L. Stojanovic

Forschungszentrum Informatik (FZI) at the University of Karlsruhe, Germany, D-76131 Karlsruhe, {maedche,motik,stojanov}@fzi.de

April 24, 2003

Keywords: Multiple and Distributed Ontologies, Ontology Evolution

1 Introduction

The Web in its' current form is an impressive success with a growing number of users and information sources. However, the growing complexity of the Web is not reflected in the current state of the Web technology. The heavy burden of accessing, extracting, interpreting and maintaining available information is left to the human user. Tim Berners-Lee, the inventor of the WWW, coined the vision of a Semantic Web in which knowledge about Web resources is represented as machine-processable metadata. Apart from this very general vision, there are many applications that may exploit and profit from a semantics-driven approach, e.g. document and content management, information integration or knowledge management, to name just a few.

Within these different application fields the usage and application of ontologies[1] is increasingly seen as key to enable semantics-driven data access and processing. When applying ontologies in these different application fields, taking into account a distributed setting like the World Wide Web, one is faced with several challenges. In this paper we consider three important ones: First of all, one has to find an appropriate representation model for ontologies, trading off between expressivity and tractability. Second, one has to be able to deal with multiple and distributed ontologies to enable reuse and interoperability. Third, one needs support in the difficult task of managing evolution of multiple and distributed ontologies.

To better understand the overall problem of managing multiple and distributed ontologies in the Web, we consider the following B2B catalog integration scenario throughout the whole article. Let's assume that some service provider A produces

[1] An ontology is a shared and machine-executable conceptual model in a specific domain of interest.

various sports utilities and that he wants to publish his catalog on-line. To allow semantics-driven access to his catalog, he needs to find an appropriate representation model for it. To enable other players in the marketplace to semantically process the catalog, he creates a sports ontology (SO) and bases his catalog on it. Let's assume further that some service provider B specializes in production of bicycles and also wants to publish his catalog on-line. Hence, he wants to create a new bicycle ontology (BO) for catalog description. When creating BO, service provider B intends to reuse as many definitions as possible from SO to speed up the engineering and to enable interoperability. However, it is not clear how to reuse these definitions in BO. Assuming that this problem is solved, after reusing SO as the basis for BO, a further problem arises when SO needs to be adapted due to a change in business requirements – it is not clear how to evolve dependent ontologies. This problem is worsened by the fact that ontologies are distributed in the Web.

In this article we present an integrated framework for managing multiple and distributed ontologies in the Semantic Web solving the above mentioned problems. Specifically, we address the following aspects: *First*, we describe a conceptual modeling framework for single, multiple and distributed ontologies, facilitating modeling and model access with a low gap between the conceptualization and its implementation. We adjust the expressiveness of traditional logic-based languages to sustain tractability. As a side effect, this makes realization of ontology-based systems using existing and well-established technologies, such as relational databases, possible. In our framework we provide features for reusing existing conceptual models. *Second*, evolution of ontologies – the timely adaptation of an ontology and consistent propagation of changes to the dependent artifacts – is a challenging management task. We present an integrated evolution process for single, multiple and distributed ontologies. *Third*, we provide an overview of a scalable implementation for managing multiple and distributed ontologies, implemented within KAON[2] – the KArlsruhe ONtology and Semantic Web infrastructure used as basis for our research and development.

Organization. This article is organized as follows. Section 2 presents our conceptual modeling approach introducing so-called OI-models as a basis for conceptual modeling in the Semantic Web. We include the definition of a mathematical model, the semantics and a comprehensive example. Additionally, we elaborate on how to deal with multiple and distributed OI-models. Section 3 is dedicated to the different aspects of evolution of single, multiple and distributed OI-models. In section 4 we present the implementation of our system. Before we conclude, we provide a comprehensive overview on related work in section 5.

2 Managing Information in OI-Models

This section introduces our conceptual modeling approach for the Semantic Web using so-called OI-models. We separate this section into three main parts. First, we introduce OI-models as a basis for conceptual modeling in the Semantic Web including the mathematical definition, the semantics and an example. Second, we

[2] http://kaon.semanticweb.org/

show how OI-models may be reused through inclusion. Finally, we discuss how reuse is achieved if OI-models are distributed across nodes in the Web.

2.1 Conceptual Modeling using OI-models

In our conceptual modeling approach we have tried to follow as closely as possible the object-oriented modeling paradigm and extend it with simple deductive features, by keeping in mind some practical aspects. Many expressive modeling languages, such as DAML+OIL [7] or OWL [9], often lack these practical considerations. This in particular relates to the support for meta-concept modeling, interpretation of domains, ranges and cardinalities, as discussed later in this subsection.

This practicality reflects itself in the system's implementation. We base our system primarily on deductive database techniques such as magic sets [3], which have proven themselves to be indispensable for achieving inferencing tractability and practicability. On the other hand, handling description logics typically requires algorithms such as tableau reasoning, which don't integrate easily with existing database infrastructure and are often intractable in practice.

Traditionally ontologies have been considered separately from their instances. However, this distinction is often blurred – some concepts have well-known instances that constitute an important part of a common vocabulary. Therefore, in our approach the information is organized in OI-models, containing both ontology entities (concepts and properties), as well as the instances. For example, an OI-model for geographical information might contain the CONTINENT concept along with its seven well-known instances. In the rest of this paper we use the terms OI-model and ontology interchangeably.

Mathematical Definition. We present our approach on an abstract, mathematical level that defines the structure of our models. We may support this structure with several different syntaxes.

Definition 1 (OI-model Structure) *An OI-model (ontology-instance-model) structure is a tuple* $OIM := (E, INC)$ *where:*

- *E is the set of entities of the OI-model,*
- *INC is the set of included OI-models.*

An OI-model represents a self-contained unit of structured information that may be reused. It consists of entities and may include a set of other OI-models (represented through the set INC). Different OI-models can talk about the same entity, so the sets of entities E of these OI-models don't need to be disjoint.

The name of an entity is often written in form ns:local_name, where ns is a shorthand for a namespace prefix. By including this prefix in the entity name, accidental name clashes are easier to avoid. The technical details how namespaces and prefixes are managed are implementation dependent (e.g. XML namespaces can be used) and are not further elaborated in this paper.

Definition 2 (Ontology Structure) *An ontology structure of an OI-model is a structure* $O(OIM) := (C, P, R, S, T, INV, H_C, H_P, domain, range, mincard, maxcard)$ *where:*

- $C \subseteq E$ *is a set of concepts,*
- $P \subseteq E$ *is a set of properties,*
- $R \subseteq P$ *is a set of relational properties (properties from the set* $A = P \setminus R$ *are called attribute properties),*
- $S \subseteq R$ *is a subset of symmetric properties,*
- $T \subseteq R$ *is a subset of transitive properties,*
- $INV \subseteq R \times R$ *is a symmetric relation that relates inverse relational properties; if* $(p_1, p_2) \in INV$*, then* p_1 *is an inverse relational property of* p_2*,*
- $H_C \subseteq C \times C$ *is an acyclic relation called concept hierarchy; if* $(c_1, c_2) \in H_C$ *then* c_1 *is a subconcept of* c_2 *and* c_2 *is a superconcept of* c_1*,*
- $H_P \subseteq P \times P$ *is an acyclic relation called property hierarchy; if* $(p_1, p_2) \in H_P$ *then* p_1 *is a subproperty of* p_2 *and* p_2 *is a superproperty of* p_1*,*
- *function domain* : $P \rightarrow 2^C$ *gives the set of domain concepts for some property* $p \in P$*,*
- *function range* : $R \rightarrow 2^C$ *gives the set of range concepts for some relational property* $p \in R$*,*
- *function* $mincard : C \times P \rightarrow N_0$ *gives the minimum cardinality for each concept-property pair,*
- *function* $maxcard : C \times P \rightarrow (N_0 \cup \{\infty\})$ *gives the maximum cardinality for each concept-property pair.*

Each OI-model has an ontology structure associated with it, consisting of concepts (to be interpreted as sets of elements) and properties (to be interpreted as relations between elements). Properties can have domain concepts and relation properties can have range concepts, which constrain the types of instances to which the properties may be applied. If these constraints are not satisfied, the ontology is inconsistent. Domain and range concepts define schema restrictions and are treated conjunctively – all of them must be fulfilled for each property instantiation. This has been done in order to maintain compatibility with various description logics dialects (e.g. OWL) which do the same. Relational properties may be marked as transitive and/or symmetric, and it is possible to say that two relational properties are inverse of each other. For each class-property pair it is possible to specify the minimum and maximum cardinalities. Concepts (properties) can be arranged in a hierarchy, as specified by the H_C (H_P) relation, whose reflexive transitive closure follows from the semantics, as defined next.

Definition 3 (Instance Pool Structure) *An instance pool associated with an OI-model is a 4-tuple* $IP(OIM) := (I, L, instconc, instprop)$ *where:*

- $I \subseteq E$ *is a set of instances,*
- L *is a set of literal values,* $L \cap E = \emptyset$*,*
- *function instconc* : $C \rightarrow 2^I$ *relates a concept with a set of its instances,*
- *function instprop* : $P \times I \rightarrow 2^{I \cup L}$ *assigns to each property-instance pair a set of instances related through given property.*

Each OI-model has an instance pool associated with it, containing instances of different concepts and property instantiations between them. Property instantiations must follow the domain and range constraints, and must obey the cardinality constraints, as specified by the semantics.

Definition 4 (Root OI-model Structure) *Root OI-model is defined as a particular, well-known OI-model with structure ROIM* := $(\{\text{KAON:ROOT}\}, \emptyset)$. KAON:ROOT *is the root concept, each other concept must subclass* KAON:ROOT *(it may do so indirectly)* [3].

[3] The prefix kaon denotes http://kaon.semanticweb.org/2001/11/kaon-lexical# namespace.

Each other OI-model must include ROIM and thus gain visibility to the root concept. Many knowledge representation languages contain the TOP concept that is a superconcept of all other concepts.

Definition 5 (Modularization Constraints) *If OI-model OIM imports some other OI-model OIM_1 (with elements marked with subscript 1), that is, if $OIM_1 \in INC(OIM)$, then following modularization constraints must be satisfied:*

- $E_1 \subseteq E, C_1 \subseteq C, P_1 \subseteq P, R_1 \subseteq R, T_1 \subseteq T, INV_1 \subseteq INV, H_{C1} \subseteq H_C, H_{P1} \subseteq H_P$,
- $\forall p \in P_1$ $domain_1(p) \subseteq domain(p)$,
- $\forall p \in P_1$ $range_1(p) \subseteq range(p)$,
- $\forall p \in P_1, \forall c \in C_1$ $mincard_1(c,p) = mincard(c,p)$,
- $\forall p \in P_1, \forall c \in C_1$ $maxcard_1(c,p) = maxcard(c,p)$,
- $I_1 \subseteq I, L_1 \subseteq L$,
- $\forall c \in C_1$ $instconc_1(c) \subseteq instconc(c)$,
- $\forall p \in P_1, i \in I_1$ $instprop_1(p,i) \subseteq instprop(p,i)$.

Cyclical inclusions aren't allowed, that is, a graph whose nodes are OI-models and whose arcs point from including to included models must not contain a cycle.

OI-model modularization is discussed in more detail in subsection 2.2.

Definition 6 (Meta-concepts and Meta-properties) *In order to allow meta-concepts, the following constraint is stated: $C \cap I$ may, but does not need to be $\emptyset$. Also, $P \cap I$ may, but does not need to be $\emptyset$.*

The consequences of meta-concept and meta-property modeling are discussed later in this subsection.

Definition 7 (Lexical OI-model Structure) *Lexical OI-model structure LOIM is a well-known OI-model with the structure presented in Figure 1*[4].

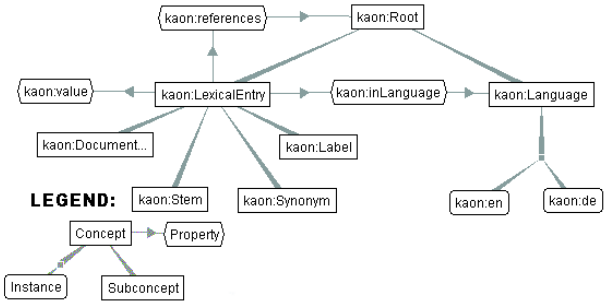

Fig. 1 Lexical OI-Model Structure

Lexical entries (instances of the KAON:LEXICALENTRY concept) reflect various lexical properties of ontology entities, such as labels, stems or textual documentation. The property KAON:REFERENCES establishes n : m relationship between lexical entries and instances. In this way, the same lexical entry may be associated with several entities (e.g. jaguar label may be associated with an instance

[4] Instead a formal definition, we present a graphical view of LOIM because we consider it to be more informative.

representing a Jaguar car or a jaguar cat). The value of the lexical entry is given by property KAON:VALUE, whereas the language of the value is specified through the KAON:INLANGUAGE property. Concept KAON:LANGUAGE represents the set of all languages, and its instances are defined by the ISO standard 639. The lexical structure is not closed – users may freely define additional subclasses of the KAON:LEXICALENTRY concept.

In order to associate lexical entries with a concept or a property, it must be viewed as an instance of the KAON:ROOT. Because of that, KAON:REFERENCES property has the concept KAON:ROOT as the domain.

Denotational Semantics. In this subsection we give meaning to OI-models by means of a denotational semantics in the spirit of description logics.

Definition 8 (OI-model Interpretation) *An interpretation of an OI-model OIM is a structure* $I = (\triangle^I, \triangle_D, E^I, L^I, C^I, P^I)$ *where:*

- $\triangle^I$ *is the set of object interpretations,*
- $\triangle_D$ *is the concrete domain for data types,* $\triangle^I \cap \triangle_D = \emptyset$*,*
- $E^I : E \rightarrow \triangle^I$ *is an entity interpretation function that maps each entity to a single element in a domain,*
- $L^I : L \rightarrow \triangle_D$ *is a literal interpretation function that maps each literal to an element of the concrete domain,*
- $C^I : \triangle^I \rightarrow 2^{\triangle^I}$ *is a concept interpretation function by treating concepts as subsets of the domain,*
- $P^I : \triangle^I \rightarrow 2^{\triangle^I \times (\triangle^I \cup \triangle_D)}$ *is a property interpretation function by treating properties as relations on the domain.*

An interpretation is a model of OIM if it satisfies the following properties:

- $C^I(E^I(\textit{kaon:Root})) = \triangle^I$,
- $\forall c, i \;\; i \in instconc(c) \Rightarrow E^I(i) \in C^I(E^I(c))$,
- $\forall c_1, c_2 \;\; (c_1, c_2) \in H_C \Rightarrow C^I(E^I(c_1)) \subseteq C^I(E^I(c_2))$,
- $\forall p, i, i_1 \;\; i_1 \in instprop(p, i) \wedge i_1 \in E \Rightarrow (E^I(i), E^I(i_1)) \in P^I(E^I(p))$,
- $\forall p, i, x \;\; x \in instprop(p, i) \wedge x \in L \Rightarrow (E^I(i), L^I(x)) \in P^I(E^I(p))$,
- $\forall p, x, y \;\; p \in R \wedge (x, y) \in P^I(E^I(p)) \Rightarrow y \in \triangle^I$,
- $\forall p, x, y \;\; p \in P \setminus R \wedge (x, y) \in P^I(E^I(p)) \Rightarrow y \in \triangle_D$,
- $\forall p_1, p_2 \;\; (p_1, p_2) \in H_P \Rightarrow P^I(E^I(p_1)) \subseteq P^I(E^I(p_2))$,
- $\forall s \;\; s \in S \Rightarrow (\forall x, y \;\; (x, y) \in P^I(E^I(s)) \Leftrightarrow (y, x) \in P^I(E^I(s)))$,
- $\forall p, ip \;\; (p, \textit{ip}) \in \textit{INV} \Rightarrow (\forall x, y \;\; (x, y) \in P^I(E^I(ip)) \Leftrightarrow (y, x) \in P^I(E^I(p)))$,
- $\forall t \;\; t \in T \Rightarrow (\forall x, y, z \;\; (x, y) \in P^I(E^I(t)) \wedge (y, z) \in P^I(E^I(t)) \Rightarrow (x, z) \in P^I(E^I(t)))$,
- $\forall p, c, i \;\; c \in \textit{domain}(p) \wedge (\exists x \;\; (E^I(i), x) \in P^I(E^I(p))) \wedge E^I(i) \notin C^I(E^I(c)) \Rightarrow$ *ontology is inconsistent,*
- $\forall p, c, i \;\; c \in \textit{range}(p) \wedge (\exists x \;\; (x, E^I(i)) \in P^I(E^I(p))) \wedge E^I(i) \notin C^I(E^I(c)) \Rightarrow$ *ontology is inconsistent,*
- $\forall p, c, i \;\; E^I(i) \in C^I(E^I(c)) \wedge \textit{mincard}(c, p) > |\{\, y \mid (E^I(i), y) \in P^I(E^I(p))\}| \Rightarrow$ *ontology is inconsistent,*
- $\forall p, c, i \;\; E^I(i) \in C^I(E^I(c)) \wedge \textit{maxcard}(c, p) < |\{\, y \mid (E^I(i), y) \in P^I(E^I(p))\}| \Rightarrow$ *ontology is inconsistent.*

OIM is unsatisfiable it is doesn't have a model. Following definitions say what can be inferred from an OI-model:

- $H_C^* \subseteq C \times C$ *is the reflexive transitive closure of the concept hierarchy if: in all models* $C^I(E^I(c_1)) \subseteq C^I(E^I(c_2)) \Leftrightarrow (c_1, c_2) \in H_C^*$,

– *$H_P^* \subseteq P \times P$ is the reflexive transitive closure of the property hierarchy if:*
in all models $P^I(E^I(p_1)) \subseteq P^I(E^I(p_2)) \Leftrightarrow (p_1, p_2) \in H_P^*$,
– *instconc** $: C \rightarrow 2^I$ represents inferred information about instances of a concept if:*
in all models $E^I(i) \in C^I(E^I(c)) \Leftrightarrow i \in$ *instconc*$^*(c)$,
– *instprop** $: P \times I \rightarrow 2^{I \cup L}$ *represents the inferred information about instances if:*
in all models $i_2 \in$ *instprop*$^*(p, i_1) \Leftrightarrow (E^I(i_1), E^I(i_2)) \in P^I(E^I(p)) \wedge$
in all models $l \in$ *instprop*$^*(p, i) \Leftrightarrow (E^I(i), L^I(l)) \in P^I(E^I(p))$,
– *domain*$^*(p) = \bigcup_{(p,p_1) \in H_P^*}$ *domain*(p_1) *denotes all domain concepts of a property,*
– *range*$^*(p) = \bigcup_{(p,p_1) \in H_P^*}$ *range*(p_1) *denotes all range concepts of a property.*

Meta-modeling. In real-world conceptual models, it is often unclear whether some element should be represented as a concept or as an instance. For example, in a semantics-driven catalog system, relationships between sports utility types and individual sports utilities can be modeled as in Figure 2. The SPORTS UTILITY concept represents the set of all types of sports utilities, with its elements being particular types of sports utilities (rafts, oars etc.) Assertions can be made about individual sport utility types (e.g. rafts are used for whitewater rafting), but each type can be viewed as the set of individual instances as well (e.g. the raft in my garage is an instance in that set). Information about sports utility types is independent from the information about particular instances (e.g. my raft can be broken). Hence, SPORTS UTILITY entity plays a dual role and can be interpreted both as a concept and as an instance. These two interpretations are connected in the image by the spanning object (the dashed line).

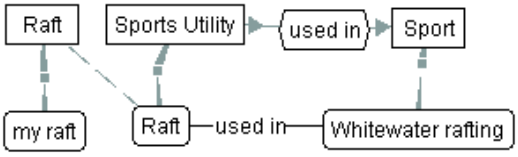

Fig. 2 Spanning Object Example

Under usual first-order semantics (employed by languages such as DAML+OIL and OWL), an element of the domain is assigned to each instance, a subset of the domain set to each concept and a relation on the domain to each property symbol. Interpreting concepts as instances is therefore not possible. To overcome these problems, we base our interpretation on HiLog [6] – a logic with the second-order syntax but first-order semantics, for which a sound and complete proof theory exists. Technically this is achieved so that the function E^I associates a domain object with each entity symbol, and functions C^I and P^I provide concept and property interpretations to domain objects.

In [32] the problems of considering concepts as instances are well explained. The proposed solution is to isolate different domains of discourse. A concept in one domain may become an instance in a higher-level domain, with these two objects being related through so called spanning object relationship. Our approach builds on that, however, without explicit isolation of domains of discourse. This has subtle consequences on how an OI-model should be interpreted. It is not allowed to ask: "What does entity e represent?" Instead, one must ask a more specific question:

"What does e represent if it is considered as either a concept, a property or an instance?"

Allowing a concept to be interpreted as an instance may cause problems. In [27] has the original RDFS semantics been criticized for its infinite meta-modeling architecture. As specified by RDFS, RDFS:CLASS is an instance of itself, which means that its interpretation must contain itself as a member. This seems dangerously close to the Russell's paradox (e.g. if RDFS is extended with negation, as in OWL, the paradox is inevitable). The same paper proposes a fixed four-layer meta-modeling architecture called RDFS(FA) introducing a strict separation between concepts and instances. Our approach doesn't suffer from these problems: the model-theoretic interpretation of the statement "A is an instance of A" is to interpret A as some domain individual α and to associate to α the concept extension containing α. Note, however, that the interpretation of A as a concept doesn't contain itself, so the Russell's paradox can't occur. We follow, however, RDFS(FA) in separating modeling primitives. This means that various relations (e.g. the RDFS:SUBCONCEPTOF property) are not available within the model, but exist in the ontology language layer. This is done to avoid ambiguities when these relations are themselves redefined (e.g. what semantics does a subproperty of RDFS:SUBCONCEPTOF have?).

Domains and Ranges. Our definition of domains and ranges differs from that of RDFS, DAML+OIL and OWL, where domains and ranges are axioms specifying sufficient conditions for class membership. For example, if a property P has concept C as the domain, then any instance I for which P has been instantiated can be classified as an instance of C. This semantics is captured by replacing the domain and range restrictions with the following conditions:

- $\forall p, c, i \;\; c \in \text{domain}(p) \wedge (\exists x \;\; (E^I(i), x) \in P^I(E^I(p))) \Rightarrow E^I(i) \in C^I(E^I(c))$,
- $\forall p, c, i \;\; c \in \text{range}(p) \wedge (\exists x \;\; (x, E^I(i)) \in P^I(E^I(p))) \Rightarrow E^I(i) \in C^I(E^I(c))$.

From our experience, while sometimes such inferencing may indeed be useful, often it is not needed, or even desired. Most users with strong background in databases and object-oriented systems, intuitively expect domains and ranges to specify the constraints on allowed ontology states. In another words, unless I is known to be an instance of C, P can't be instantiated for I in the first place. This has the following benefits: First, treating domains and ranges as constraints makes it possible to guide the user in the process of providing information about instances. If domains and ranges are treated as axioms, any property can be applied to any instance, making it difficult to constrain user's input. Second, similar problems occur when evolving the ontology. E.g., if I is removed from the extension of C, it can be computed that the instance of P for I must be removed as well. If domains and ranges are axioms, however, it is not clear how to change the ontology so that it still makes sense. Third, treating domains and ranges as axioms introduces significant performance overhead in query answering. For example, to compute the extension of some concept, one must classify instances according to the domain and range axioms. Therefore, if only the constraint semantics is needed, the system will suffer from unnecessary performance overhead.

Cardinalities. In our approach we treat cardinalities as constraints regulating the number of property instances that may be specified for instances of each concept.

This is different from OWL and other description logic languages, where cardinalities are axioms specifying that instances with particular number of property instances can be inferred to be instances of some concept. Similar arguments as in the case of domain and range semantics apply here as well.

Example. Figure 3 graphically presents a simple OI-model describing various types of sports, sports utilities and their relationships in a semantical on-line catalog, as described in the scenario presented in section 1. On the right-hand side there is a hierarchy of various sports. The concept SPORT represents a set of all sports, which can be divided into team and individual sports, as well as outdoor and indoor sports. A particular sport can be classified under several different parent concepts, thus reflecting various aspects of sports. For example, WHITEWATER RAFTING is an instance of TEAM SPORT and OUTDOOR SPORT concepts.

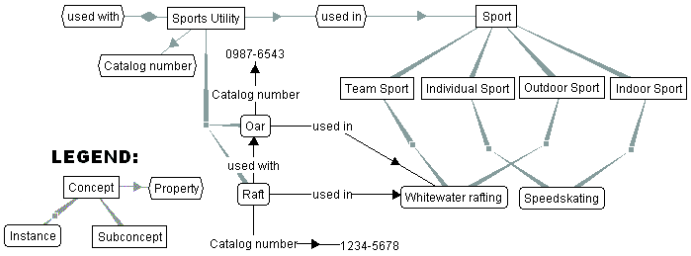

Fig. 3 Example Domain OI-model

This can be interpreted as TEAM SPORT being a set of all team sports, and WHITEWATER RAFTING being a member in this set. Sports utilities are related to sports that they are used in. Further, some sports utilities are used together (e.g. OAR and RAFT). This information is represented through USED-WITH property. This property is, by its nature, transitive and symmetric. Such information can be used by the company hosting the catalog to suggest possible items of interest to the buyer. Each sports utility has a CATALOG NUMBER associated with it. This is a property that has a literal value whose semantics is not further specified.

2.2 Multiple OI-Models

In traditional software systems significant attention is devoted to keeping modules well separated and coherent with respect to functionality, thus making sure that changes in the system are localized to a handful of modules. Reuse is seen as the key method in reaching that goal, striving to completely eliminate the copy-and-paste reuse – the prominent source of problems on software projects. Ontology-based systems in the Web are just a special class of software systems, so the same principles apply. If reuse is performed through duplication, problems arise when the reused ontology changes, as these changes must be replayed on various multiple copies. Paraphrasing the open-closed reuse principle [21], each ontology should be a closed, consistent and a self-contained entity, but open to extensions in other ontologies. These goals may be achieved by incorporating an explicit mechanism for including ontologies by reference into ontology languages and tools.

According to definition 5, reuse is supported by allowing an OI-model to include other OI-models, thus obtaining the union of the definitions from all included models. In our approach cyclical inclusions are not allowed because evolution of cyclically dependent OI-models would be too difficult. Inclusion is performed by-reference – models are virtually merged, however, the information about the origin of each entity is represented explicitly.

Figure 4 presents four example OI-models (SO – sports ontology, BO – bicycle ontology, CO – climbing ontology and ICO – integrated catalog ontology). BO and CO each include SO, thus gaining immediate access to all of its definitions. However, the information about the origin of ontology entities retained. Thus, the following distinctions may be made:

- In SO and CO SPORTS UTILITY concept doesn't have any sub- or superconcepts. However, in BO it has one subconcept BICYCLE, and in ICO it has one subconcept and one superconcept CATALOG ITEM.
- Relationships between concepts also belong to appropriate OI-models. Hence, it is possible to determine that SPORTS UTILITY is made a subconcept of CATALOG ITEM in ICO.
- In SO the property USED IN has only SPORTS UTILITY as domain concept, whereas in ICO it has an additional domain concept POWER DRINK.

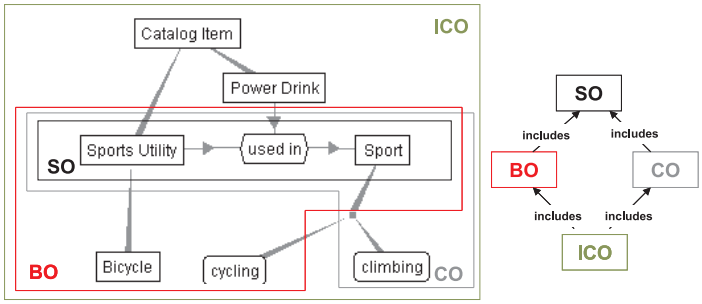

Fig. 4 OI-model Inclusion

On the right-hand side the direct acyclic inclusion graph between OI-models is shown. SO is indirectly included in ICO twice (once through BO and once through CO). However, ICO will contain all elements from SO only once (e.g. in ICO there will be only one SPORTS UTILITY concept). The possibility of including an OI-model through multiple paths has significant consequences on the ontology evolution, as discussed in subsection 3.2.

Our approach is currently limited to including entire models, rather than including subsets. Also, when a model is reused, information can only be added, not retracted. Although such advanced features may sometimes be useful, we deliberately limit our approach. It is much more difficult to ensure the consistency of the including model if only part of some model is included, since it is not clear which subset of elements to include. For example, if USED IN property is not included in ICO, it is not clear how to treat instances using this property. Further, changing ontologies becomes more complex, because it is not clear how to propagate changes in ICO to SO. Finally, we don't support resolving semantical heterogeneities between included models (e.g. establishing equivalences between the BICYCLE and

FAHRRAD concepts) – we plan to extend our approach to handle such cases in future.

2.3 Distributed OI-Models

Ontology inclusion allows reusing OI-models available within one node (server) in the system. However, we envisage the Semantic Web where OI-models are spread across many different nodes, so the inclusion mechanisms cannot be used directly. There are two possible solutions how to achieve reuse in this case.

The first solution is to make all OI-models accessible through an ontology server, which could integrate the information from included OI-models virtually (on-the-fly) by accessing the servers of these OI-models. Such solution has the benefit that all changes in the included OI-models are immediately visible in the including OI-models. While this desirable feature increases the consistency, it has several serious drawbacks:

- Servers are tightly coupled – a failure of one system will cause failure of all servers that include the OI-model.
- Standard top-level ontologies will be reused in many ontologies. Servers hosting them will therefore be overloaded, because they will often be contacted by many other servers.
- Because answering every query requires distributed processing, the performance of the system with today's infrastructure would be unacceptable.

Therefore, a more practical solution to the problem in the WWW context is to replicate distributed OI-models locally and to include them in other OI-models. Replication eliminates afore mentioned problems, but introduces significant evolution and consistency problems, further discussed in section 3. The most important constraint is that replicated OI-models should never be modified directly. Instead, the modification should always be performed at the source and changes propagated to replicas using the distributed evolution process.

Returning to the example from the section 1, Figure 5 shows a network of three service providers creating their ontologies. SO and CO are defined at the server of service provider A. Since CO is defined at the same node as the SO, no replication is necessary. BO is defined at the server of the service provider B, so to reuse SO, it must be replicated to his server. Finally, ICO is defined at the server of service provider C, so SO, BO and CO must be replicated to his server.

In order to replicate an OI-model, it must be physically accessed. OI-models on the Web are typically known under a well-known URI, which can be used to access the OI-model through appropriate protocol (e.g. HTTP). However, this introduces problems when the OI-model is replicated, since the URI used to access the OI-model and the URI under which the OI-model is originally known become different. To consistently handle this, we associate two different URIs with each OI-model:

- The logical URI is unique for each OI-model and is always the same, regardless of the OI-model's location. The uniqueness of the URI is typically achieved by incorporating the Internet name of the organization that created the OI-model.

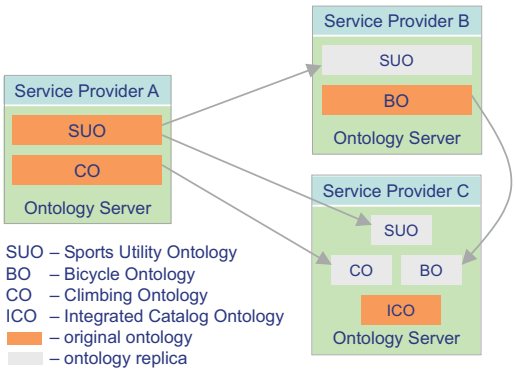

Fig. 5 Distributed OI-Models

– The physical URI unambiguously identifies the location of the OI-model and contains all information necessary to access the OI-model, such as the protocol to be used or relevant connection parameters.

For example, the SO from our example may have the logical URI http://www.sport.com/so. No other OI-model with that URI exists anywhere in the world. However, the OI-model may be replicated to the file system, and the physical URI will be file:/c:/so.kaon. If the OI-model is stored in the database, then its physical URI may be jboss://wim.fzi.de:1099?http://www.sport.com/so.

3 Evolution

Ontology evolution can be defined as the timely adaptation of an ontology and a consistent propagation of changes to the dependent artifacts. The complexity of ontology evolution increases as ontologies grow in size, so a structured ontology evolution process is required. Such a process has been described in [18]. The process starts with capturing changes either from explicit requirements or from the result of change discovery methods. Next, changes are represented formally and explicitly. The semantics of change phase prevents inconsistencies by computing additional changes that guarantee the transition of the ontology into a consistent state. In the change propagation phase all dependent artifacts (ontology instances on the Web, dependent ontologies and application programs using the changed ontology) are updated. During the change implementation phase required and induced changes are applied to the ontology in a transactional manner. In the change validation phase the user evaluates the results and restarts the cycle if necessary.

In this paper we extend this process towards multiple, distributed ontologies. As shown in Table 1, two dimensions of the overall ontology evolution problem may be identified.

The first dimension defines the number of ontologies being evolved, whereas the second specifies their physical location. Since it is not possible to fragment [25] one ontology across many nodes, we discuss ontology evolution at three levels only. In subsection 3.1 we summarize the single ontology evolution problem. In subsection 3.2 we extend the change propagation and capturing phases to cover

		Nodes	
		One	Multiple
Ontologies	One	Single OE	-
	Multiple	Dependent OE	Distributed OE

Table 1 Levels of Ontology Evolution (OE) Problem

the evolution of multiple dependent ontologies within a single node. Finally, in subsection 3.3 we extend the change capturing and change implementation phases of the dependent evolution process to support evolution of distributed ontologies.

3.1 Single OI-Model Evolution

For evolution of single ontologies the essential phase is the semantics of change phase, whose task is to maintain ontology consistency. Applying elementary ontology changes [30] alone will not always leave the ontology in a consistent state. For example, deleting a concept will cause subconcepts, some properties and instances to be inconsistent.

Definition 9 (Single Ontology Consistency) *A single ontology is consistent if it satisfies a set of invariants defined in the ontology model from the subsection 2.1 and if all used entities are defined.*

Returning to the example in Figure 3, if SPORT concept is deleted, its subconcepts would be inconsistent, since the parent concept is not defined any more. To prevent inconsistencies, all subconcepts (INDOOR SPORT, OUTDOOR SPORT, TEAM SPORT and INDIVIDUAL SPORT) have to be deleted as well. Moreover, the removal of concepts causes the removal of their instances (e.g. the SPEEDSKATING instance of the INDOOR SPORT concept) and the removal of lexical information. Further, since the removal of a concept which is in the range of some property results in syntax inconsistency, before the SPORT concept is deleted, it must be removed from the range of the USED IN property. However, properties without range concepts are not allowed, so the property must be deleted as well. To do that, the SPORTS UTILITY concept must be removed from the domain. The complete list of necessary changes obtained in the semantics of change phase is presented in Figure 6.

However, there are many ways to achieve consistency after a change request. For example, when a concept from the middle of the hierarchy is being deleted, all subconcepts may either be deleted or reconnected to other concepts. If subconcepts are preserved, then properties of the deleted concept may be propagated, its instances distributed, etc. Thus, for some change in the ontology, it is possible to generate different sets of additional changes, leading to different final consistent states. Further, the consistent state may be defined in multiple ways. For example, properties without domain and/or range may or may not be considered inconsistent. Most of existing systems for the ontology development and management provide only one possibility for realizing a change and this is usually the simplest one. For example, the deletion of a concept always causes the deletion of all its subconcepts. To overcome this, a mechanism is required for users to manage changes resulting not in an arbitrary consistent state, but in a consistent state

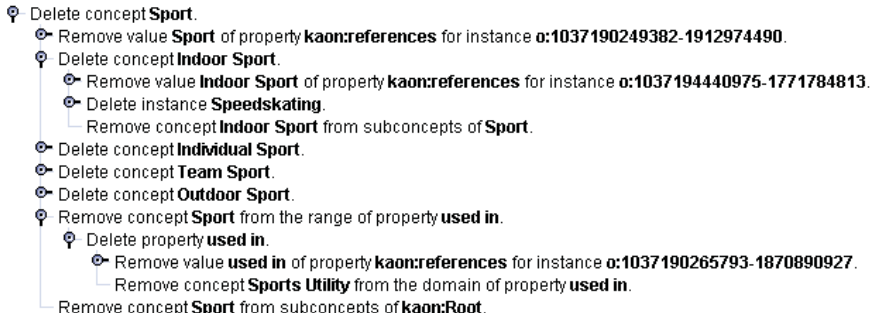

Fig. 6 Generated Changes

fulfilling the user's preferences. We introduce the concept of an evolution strategy encapsulating policy for evolution with respect to user's requirements. To resolve a change, the evolution process needs to determine answers at many **resolution points** - branch points during change resolution where taking a different path will produce different results. We have identified the following set of resolution points:

- how to handle orphaned concepts - those concepts that don't have parents any more;
- how to handle orphaned properties - those properties that don't have parents any more;
- how to propagate properties to the concept whose parent changes;
- what constitutes a valid domain of a property;
- what constitutes a valid range of a property;
- whether a domain (range) of a property can contain a concept that is at the same time a subconcept of some other domain (range) concept;
- the allowed shape of the concept hierarchy (i.e. multiple paths to a superconcept);
- the allowed shape of the property hierarchy; (i.e. multiple paths to a superproperty);
- if instances must be consistent with the ontology.

Each possible answer at each resolution point is an **elementary evolution strategy**. For example, in case of the first issue, orphaned subconcepts of a concept may be connected to the parent concept(s) of that concept, connected to the root concept of the hierarchy or deleted as well. Common policy consisting of a set of elementary evolution strategies, each giving an answer for one resolution point, is an **evolution strategy**. Thus, an evolution strategy unambiguously defines the way how elementary changes will be resolved [30]. A particular evolution strategy is typically chosen by the user at the start of the ontology evolution process. Let's assume that the chosen evolution strategy determines that orphaned concepts should be reconnected to the root concept and that a property can exist without a range concept. The list of generated changes for the same request (the removal of a concept SPORT from the example in Figure 3) is shown in Figure 7. This list is quite different from the changes shown in Figure 6, since the selected evolution strategies are different.

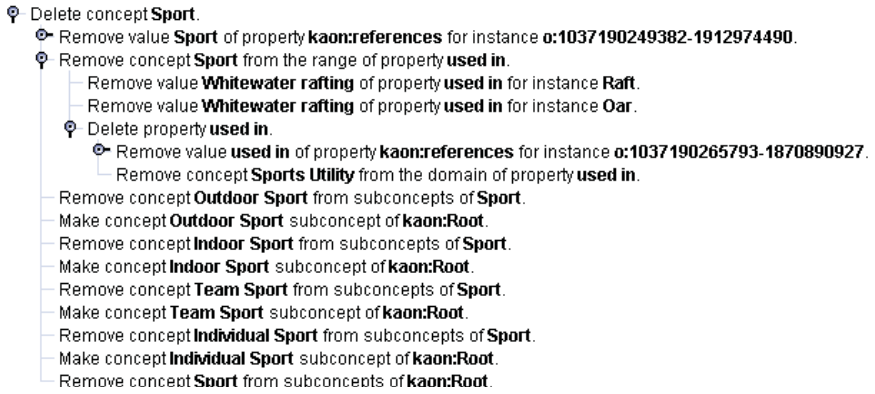

Fig. 7 Generated Changes based on the explicitly selected Evolution Strategy

3.2 Multiple OI-Model Evolution

In this subsection we extend the single ontology evolution approach to take into account the inclusion relationships between ontologies within one node. However, we still consider evolution of ontologies within one node only and extend this to the distributed setting in the following subsection. An ontology that includes other ontologies is called the dependent ontology. As the included ontology is changed, the consistency of the dependent ontology may be invalidated.

Definition 10 (Dependent Ontology Consistency) *A dependent ontology is consistent if the ontology itself and all its included ontologies, observed alone and independently of the ontologies in which are reused, are single ontology consistent.*

Returning to the example of Figure 4, if the SPORTS UTILITY concept from SO is deleted, the ontology BO, and through transitivity of inclusion the ICO as well, will be inconsistent, since the BICYCLE and CATALOG ITEM concepts will have a parent concept and a child concept respectively, that are not defined. Moreover, it is important to notice that applying the deletion of the SPORTS UTILITY concept to the outer-most ontology (ICO) only is not sufficient. In ICO the USED IN property has two domain concepts, so removing one of them will not trigger the removal of the property. Therefore, if SO is considered independently, it is inconsistent, since the USED IN property will have no domain concept in this ontology.

This example shows that maintaining consistency of a single ontology is not sufficient; dependent ontology consistency must be taken into account as well. This may be achieved by propagating changes from the changed ontology to all ontologies that include it. There are two ways of doing that [4]:

- *Push-based approach*: Changes from the changed ontology are propagated to dependent ontologies as they happen.
- *Pull-based approach*: Changes from the changed ontology are propagated to dependent ontologies only at their explicit request.

The pull-based approach is better suited for less stringent consistency requirements. Using this approach dependent ontologies may be temporarily inconsistent.

This makes recovering the consistency of dependent ontologies difficult, since the information about the original state of the changed ontology is lost. For example, when the concept SPORTS UTILITY is deleted, its position in the concept hierarchy is lost and is not available when resolving inconsistencies of the BICYCLE concept in BO.

The push-based approach is suitable when strict dependent ontology consistency is required, since the information about the original state of the changed ontology is available for the evolution of the dependent ontology. For example, the removal of the concept SPORTS UTILITY requires previous resolution of the consistency of the BICYCLE concept in BO. We choose to take this approach since in our target applications the permanent consistency of ontologies within one node is of paramount importance.

By adopting the push-based approach, there are three different strategies for choosing the moment when changes are propagated [28]. Using the periodic delivery, changes are propagated at regular intervals. Using ad-hoc delivery, changes are not propagated according to a previously defined plan. Both of these strategies are unacceptable for dependent ontology evolution, since they cause temporal inconsistencies of dependent ontologies. Therefore, we propagate changes immediately, as they occur.

We incorporate the push-based approach by extending the change propagation and change capturing phases of the single ontology evolution process as shown in Figure 8.

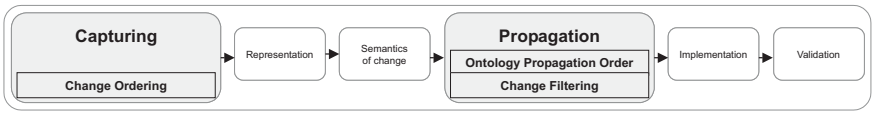

Fig. 8 Dependent Ontology Evolution Process

The role of the Ontology Propagation Order component is to determine to which dependent ontologies and in which order should the changes be propagated. The role of the Change Filtering component is to determine which changes must be propagated to which ontologies. The Change Ordering component determines the order in which changes must be received by each ontology.

Ontology Propagation Order. When propagating changes between dependent ontologies on a single node, the following three aspects relating to the ontology propagation order must be considered:

- As changes occur in an ontology, they must be pushed to all ontologies that either directly or indirectly (through other ontologies) include the changed ontology.
- In order to propagate a change to an ontology, the change must previously be processed by all ontologies included in the target ontology. Therefore, all ontologies on a single node are topologically ordered[5] according to their inclusion relationship. The topological order organizes the dependent ontologies in such a way that for each O1 and O2, if O1 includes O2 directly or indirectly, then O2 occurs before O1 in the linear ordering.

[5] The topological order of a directed graph is an ordering of graph's nodes where each node occurs after all of its predecessors.

– Since all ontologies at a node are topologically ordered, when changes are propagated to dependent ontologies, only those ontologies that include the changed ontology and that follow the changed ontology in the topological order must be visited. Note that if cyclical inclusions of OI-models were allowed, the propagation order would contain cycles and would be extremely hard to manage.

Returning to the example in Figure 4, changes in SO must be propagated to BO, CO and ICO (since ICO includes SO indirectly though BO and CO). Further, several topological orders may exist (SO, BO, CO, ICO or SO, CO, BO, ICO), since some ontologies are independent of each other (e.g. BO and CO). The propagation of changes must be performed in either one of these orders. Assuming the first topological order (SO, BO, CO and ICO), a change in BO is propagated only to ICO – although CO is after BO in the topological sort, it doesn't include BO so it doesn't receive BO's changes.

Change Filtering. As a change from the source ontology S is propagated to a dependent ontology D, to maintain the consistency of D additional changes will be generated as explained in subsection 3.1. These changes must also be propagated further up the ontology inclusion topological order. However, only induced changes should be forwarded. If original changes were propagated as well, then ontologies that include D would receive the same change multiple times: directly from S and indirectly from all ontologies on any inclusion path between D and S. This would result in an invalid ontology evolution process, since the same change cannot be processed twice. In order to prevent that, propagated changes are filtered.

Returning to the example in Figure 4, deletion of the SPORTS UTILITY concept in SO is propagated to BO resulting in new changes: the removal of the BICYCLE concept as the subconcept of the SPORTS UTILITY concept and the removal of the BICYCLE concept itself (if the evolution strategy requires the removal of the orphaned concepts). Only these two changes are propagated to ICO. Removal of the concept SPORTS UTILITY is propagated to ICO from SO directly and must not be propagated from BO. Notice that change filtering is not done for the sake of performance: if SPORTS UTILITY were propagated to ICO from BO as well, then ICO would receive the same change twice, and the second change would fail, since the concept has already been deleted.

Change Ordering. The order of processing changes in each ontology is important. Let's assume that S is the ontology being changed, I is some ontology that directly includes S and D is some ontology that directly includes I. It is important that D processes changes generated by I before changes generated by S. Otherwise, if D receives changes from S before changes from I, S's changes will generate additional changes in D that include those that will later be received from I. This in turn will also lead to processing the same change twice. This approach is recursively applied when D and S are connected with paths of length greater than two. Returning to the example in Figure 4, ICO should process the removal of the SPORTS UTILITY concept after processing the removal of the subconcept BICYCLE from BO. If this were not the case, processing removal of SPORTS UTILITY in ICO would generate removal of the subconcept BICYCLE in ICO, which will then be later received from BO.

The algorithm for evolution between dependent ontologies within one node is presented in Algorithm 1.

Algorithm 1 Dependent Ontology Evolution Algorithm

```
EVOLVEONTOLOGIES(ℒ𝒞, o)
Require: ℒ𝒞 - list of changes, o - ontology being changed
 1: for all c ∈ ℒ𝒞 do
 2:     PROCESSCHANGE(c, o)
 3: end for
PROCESSCHANGE(c, o)
Require: c - change to process, o - ontology being changed
 4: 𝒯𝒮 = topological sort of ontologies at the node
 5: es = evolution strategy for o
 6: /*Semantics of Change*/
 7: while generated change gc by es for c in o do
 8:     processChange(gc, o)
 9: end while
10: /*Change Filtering*/
11: if c is generated in o then
12:     /*Ontology Propagation Order*/
13:     for all ontology d after o in 𝒯𝒮 do
14:         if ontology d includes o then
15:             /*Change Ordering*/
16:             processChange(c, d)
17:         end if
18:     end for
19: end if
20: /*Change Implementation*/
21: change ontology o according to c
```

It processes all changes that are requested by the user through the procedure PROCESSCHANGE (cf. 1–3). This procedure resolves a change by generating the additional changes needed to keep the consistency of the ontology *o* for which the method was called (cf. 7–9). Only changes generated in *o* are propagated (cf. 11) to the all ontologies including *o* according to the topological order of all ontologies within the node (cf. 13–14). The recursive call (cf. 16) to the PROCESSCHANGE procedure for the filtered change and topological order of dependent ontologies guarantees that the receiving ontologies will process the changes from the directly included ontologies before changes from the indirectly included ontologies. Finally, the change is applied to the ontology *o* (cf. 21).

3.3 Distributed OI-Model Evolution

A distributed dependent ontology is an ontology that depends on an ontology residing at a different node on the network. The physical distribution of ontologies is very important, since it creates additional problems that are not encountered when the ontologies are collocated. This additional complexity stems from the fact that reusing distributed ontologies is achieved through replication (see section 2.3). Since the original ontology is updated autonomously and independently of replicas, this in turn introduces an additional type of consistency.

Definition 11 (Replication Ontology Consistency) *An ontology is replication consistent if it is equivalent to its original and all its included ontologies (directly and indirectly) are replication consistent.*

To explain this notion, we assume a distributed system of replicated ontologies as shown in Figure 5. Ontology SO at service provider B is replication inconsistent if it hasn't been updated according to changes in its original at the service provider A. This implies the replication inconsistency of BO at provider B (since BO includes SO which is replication inconsistent). Finally, this implies the replication inconsistency of ICO at the service provider C in the same way. To resolve replication inconsistencies between ontologies, first a way of synchronizing distributed ontologies is needed. Table 2 discusses the pros and cons of two well-known approaches [4] for synchronizing distributed systems. Although seemingly similar, there is significant difference to the approaches described in subsection 3.2, as this case deals with a distributed system.

	Push	Pull
Dependency Information	centralized	local
Complexity of management	high	medium
Type of consistency	strict	loose
Communication overhead	high	optimized

Table 2 Push vs. Pull Synchronization of Ontologies

Under push synchronization the changes of originals are propagated to ontologies including replicas immediately. We identify several drawbacks of using this approach for realistic scenarios on the Web. First, to propagate changes, for each ontology one must know which ontologies reuse it. Thus, an additional centralized component managing inclusion dependencies between ontologies is needed. Second, with the increase in the number of ontologies and their reuse, the number of dependencies will grow dramatically. Managing them centrally will be too expensive and impractical, since the problem of evolving dependencies emerges. Third, forcing all ontologies to be "strictly" consistent at all times reduces the possibility to express diversities in a huge space such as the Web. Subjects on the Web may not be ready to update their dependent ontologies immediately and may opt to keep the older version deliberately. Finally, the changes are propagated one-by-one, introducing significant communication overhead. Grouping changes and sending them on demand will perform better.

Therefore, in the distributed environment we advocate using the pull synchronization. Under this approach information about included ontologies is stored in the dependent ontology, thus eliminating the need for central dependency management. Original ontologies are checked periodically to detect changes and collect deltas. During this process, it may be possible to analyze changes and to reject them if they don't match current needs. Thus, we propose a "loosely" consistent system, since replication consistencies are enforced at request. Permitting temporary inconsistencies is a common method of increasing performance in distributed systems [25].

Hence, we use the pull approach for synchronizing originals and replicas, whereas we use the push approach for maintaining consistency of ontologies within

one node. Thus, our solution employs a hybrid synchronization strategy combining their favorable features while avoiding their disadvantages.

Regardless of the synchronization approach, the question how replication inconsistencies are actually resolved is raised. We note that replication inconsistencies cannot be resolved by simply replacing the replica with the new version of the original. This will cause inconsistencies of the dependent ontologies, as discussed in subsection 3.2. Instead, replication and dependency inconsistency must be resolved together in one step. This can be achieved by applying dependent evolution algorithms on deltas – changes that have been applied to the original since the last synchronization of the replica. By using the pull synchronization strategy and by applying the dependent evolution process from Figure 8 to deltas, we derive the distributed ontology evolution process through three extensions.

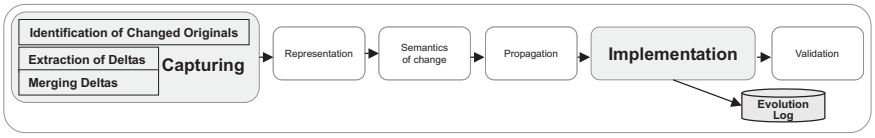

Fig. 9 Distributed Ontology Evolution Process

This process, shown in Figure 9, is responsible for propagating changes from originals to replicas. We extend the implementation phase by introducing the evolution log for keeping information about performed changes. Further, we extend the change capturing phase by three components. During identification of changed originals we identify which original ontologies have changed. In extraction of deltas we identify the changes performed at the original and not at the replica by reading the evolution log. Finally, during merging of deltas we generate a cumulative list of changes that must be performed at the replica.

3.3.1 Logging Changes

In order to resolve replication inconsistencies, two known ways of identifying deltas between originals and replicas are known [25]: (1) the full content of the original ontology may be compared to the replica; (2) the history of changes to the original may be kept explicit. The first solution requires extracting changes from differences between the original and the replica, which is a complicated and a time-consuming process. Further, to compare ontologies, the current version of the original must be copied temporarily to the replica's node. This may incur unnecessary communication overhead. If a concept is added to a large ontology, it is better to transfer only the information about this addition, instead of transferring the whole ontology.

To avoid these drawbacks, we follow for the second option. For each distributed ontology an instance of a special evolution log ontology is created, which tracks the history of changes to the ontology. Apart from the distributed evolution, the evolution log is also used to provide the following capabilities:

– Users often want to undo the changes to the ontology. For each elementary change a sequence of inverse changes may be derived that completely undo the original changes. Hence, by applying inverse changes in reverse order any previous state of an ontology may be reconstructed [30].

– With each change additional meta-information may be associated. This information can serve as a source for different knowledge discovery methods, e.g. mining about change trends.

An evolution log ontology models what changes, why, when, by whom and how are performed in an ontology. Each change is represented as an instance of one of the subconcepts of the concept CHANGE. The structure of the hierarchy of change types reflects the underlying ontology model by including all possible types of changes (e.g. ADDENTITY, REMOVEENTITY etc.). Additional information, such as the date and time of change, as well as the identity of the change initiator, may be associated through appropriate properties. Information enabling decision-making support, such as cost, priority, textual description of the reason for change etc. may also be included. Entities from the ontology being changed are related to instances of the CHANGE concept through HAS_REFERENCEENTITY property. As described previously, elementary changes may cause new changes to be introduced by the evolution strategy in order to keep the ontology consistent – such dependencies may be represented using the CAUSECHANGE property. Groups of changes of one request are maintained in a linked list using the HAS_PREVIOUSCHANGE property.

3.3.2 Resolving Replication Inconsistencies

As shown in Figure 9, resolving replication inconsistencies is performed through three additional components. Resolving replication inconsistencies is initiated by specifying an original whose included replicas should be updated and is performed as follows.

Identification of Changed Originals. This step first checks whether resolution of replication inconsistency can be performed at all. If for some directly included replica the original has replication inconsistency, this step is aborted. Otherwise, a list of directly included replicas that have pending replication inconsistency (but whose original is replication consistent) is determined. Since the dependent ontology consistency for the ontologies on the same node is required, this approach is recursively applied on all ontologies that include the ontology whose replication inconsistency is resolved.

Let's assume that the service provider C from Figure 5 wants to resolve the replication inconsistency of ICO. Its directly included replicas, namely BO and CO, are examined. For each of them the replication consistency of the original is checked. If BO at the service provider B has replication inconsistency (due to changes from A in SO which haven't been applied at B's replica of SO), then the process is aborted. If BO at the service provider B is replication consistent, but BO at service provider C is not (since BO at B has been changed), then BO is identified for further analysis. The consistency of the BO's original is required since ICO will obtain changes from SO through BO's and CO's evolution log. In order to optimize this step, the set of directly included ontologies to be taken into account may be reduced by eliminating all directly included ontologies that are available through some other paths. In the case that the ontology ICO directly includes the ontology SO, the ontology SO would be eliminated from the further consideration, since it can be obtained through the ontologies BO and CO. Replication consistency is performed by determining the equivalence of the ontology with its original and

by recursively determining the replication consistency of included ontologies. The following information is needed to perform that:

- Each ontology must contain a physical URI of its original.
- Each ontology must contain a physical URI of its evolution log.
- Each ontology has a version number associated with it that is incremented each time when an ontology is changed. Thus, checking the equivalence of the replica and the original can be done by simple comparison of that number.

Extraction of Deltas. After determining directly included replicas to be updated, the evolution log for these ontologies is accessed. The location of the evolution log is specified within each ontology and is copied to replicas. For each log the extracted deltas contain all changes that have been applied to the original after the last update of the replica, as determined by the version numbers.

Merging Deltas. Deltas extracted from evolution logs in the previous step are merged into a unified list of changes. Since an ontology can be included in many other ontologies, its changes will be included into evolution logs of all of these ontologies. Hence, the merge process must eliminate duplicates. Also, changes from different deltas caused by the same change from a common included ontology should be grouped together. For example, if the ontology SO is changed, the evolution logs of the BO and CO will contain these changes, as well as their own extensions. Hence, when changes from BO's and CO's logs are merged in order to update ICO, the changes to SO will be mentioned twice. Thus, only one change to SO should be kept while discarding all others. However, changes in BO and CO caused by a change in SO must be grouped together.

Algorithm 2 Distributed Ontology Evolution Algorithm

```
UPDATEDISTRIBUTEDONTOLOGY(o)
Require: o - ontology that have to be updated
 1: /*Identification Of Changed Originals*/
 2: inconsistentReplicas=identificationOfChangedOriginals(o)
 3: for all inconsistentReplica in inconsistentReplicas do
 4:    /*Extraction of Deltas*/
 5:    evolutionLog=findEvolutionLog(inconsistentReplica)
 6:    deltas=readEvolutionLog(evolutionLog)
 7:    /*Merging Deltas*/
 8:    changes=mergeDeltas(deltas)
 9: end for
10: evolveOntologies(changes,o)
IDENTIFICATIONOFCHANGEDORIGINALS(o)
Require: o - ontology that have to be updated
11: replicas=findFirtsLevelReplicas(o)
12: for all replica in replicas do
13:    includedOntologies=findAllIncludedOIModels(replica)
14:    for all includedOntology in includedOntologies do
15:       if includedOntology is not replication consistent then
16:          generate exception("Included models are not updated yet!")
17:       end if
18:    end for
19: end for
```

The algorithm of our approach for evolution between distributed ontologies is presented in Algorithm 2. It starts with the identification of changed originals (cf. 2). This includes the checking whether the evolution can be performed at all and returns all included replicas that are out-of-date (cf. 11–19). For each of these replicas, the evolution log is accessed in order to extracts deltas (cf. 5–6). These changes are merged with the changes from the other evolution logs (cf. 8). Finally, this integrated list of deltas from all out-of-date replicas is processed using the dependent ontology evolution process (cf. 10) as discussed in subsection 3.2.

4 Implementation

KAON is an open-source ontology management infrastructure targeted for semantics-driven business applications. Important focus of KAON is on integrating traditional technologies for ontology management and application with those used typically in business applications. The KAON architecture is presented in Figure 10.

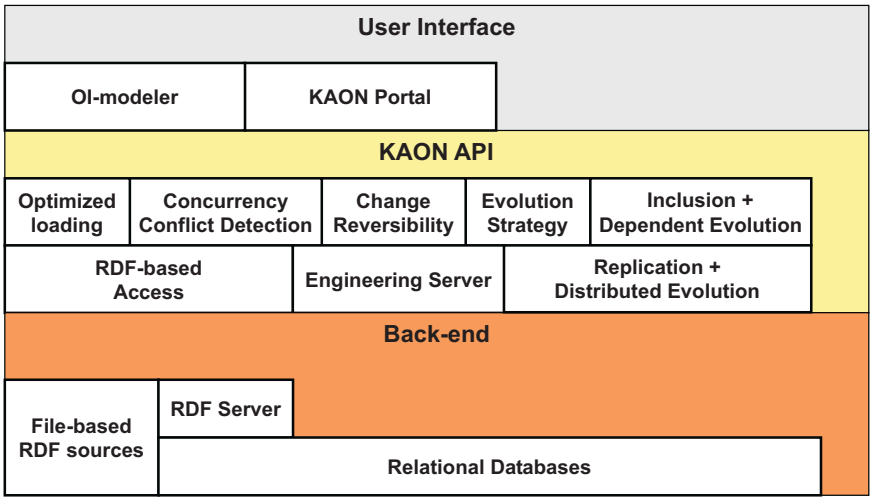

Fig. 10 KAON Implementation Architecture

The focal point of the KAON architecture is its ontology API (KAON API), consisting of a set of interfaces for access to ontology entities. For example, there are Concept, Property and Instance interfaces, containing methods for accessing ontology concepts, properties and instances, respectively. The API incorporates some other important elements required for ontology management:

- Optimized loading component is responsible for bulk-loading of ontology entities. To improve performance, entities are cached at the client.
- Concurrency conflict detection is responsible for detecting and resolving conflicts resulting in concurrent updates of different users. For example, if one user updates the ontology, then other active users must be notified of this update. Alternatively, if a user attempts to update the ontology using stale information, the conflict must be detected.

- Change reversibility is responsible for keeping track of the ontology changes in an evolution log in order to be able to reverse them at user's request. Further, the evolution log is also used by the distributed ontology evolution.
- Evolution strategy is responsible for making sure that all changes applied to the ontology leave the ontology in a consistent state and for preventing illegal changes. Also the evolution strategy allows the user to customize the evolution process.
- Ontology inclusion facilities together with dependent evolution are responsible for managing multiple ontologies within one node.
- Ontology replication facilities together with distributed evolution are responsible for enabling reuse of distributed ontologies.

One implementation of the KAON API is based on the RDF API, and thus allows access to RDF repositories. Although it offers capabilities for accessing remote RDF repositories, such as RDF Server, it is primarily used for management of local RDF ontologies stored as files. Engineering Server is an implementation of the API directly based on relational databases and is targeted for cases where concurrent access is needed. The name Engineering Server stems from the fact that the database schema is optimized for ontology engineering, during which adding and deleting concepts are frequent operations that must be done transactionally. Therefore, the engineering server uses a fixed number of tables, rather than a table per concept. The server has been heavily optimized and tested on an ontology consisting of 100,000 concepts, 66,000 properties and 1,000,000 instances, where loading related information about 20 ontology entities takes under 3 seconds, while deleting a concept in the middle of the concept hierarchy takes under 5 seconds. This informal test has been conducted on a usual single processor desktop computer running Windows XP with 256MB of RAM.

To implement light-weight inferences, KAON relies on datalog to provide desired semantics. Datalog queries are evaluated bottom-up. This evaluation strategy manages information in sets, matching well with the way how queries are performed in relational databases. In order to limit the amount of computation only to facts relevant to the query, magic sets transformation technique [3] is applied. Briefly, this technique simulates top-down binding propagation through bottom-up computation.

On top of KAON API various applications have been realized. KAON Portal is a tool for building ontology-based Web sites. Furthermore, within KAON we have developed OI-modeler, an end-user application that realizes a graph-based user interface for single and distributed OI-model creation and evolution [18]. Figure 11 shows the graph-based view of an OI-model.

OI-modeler supports ontology evolution at the user level (see the right side of the screenshot). The figure shows a modeling session where the user attempted to remove the concept SPORTS UTILITY. Before applying this change to the ontology, the system computed the set of additional changes that must be applied. The tree of dependent changes is presented to the user, thus allowing the user to comprehend the effects of the change before it is actually applied. Only when the user agrees, the changes will be applied to the ontology. OI-modeler supports working with multiple ontologies at the same time. In the upper right-hand corner of this picture one can see a graph showing the inclusion dependencies among all open OI-models. By selecting a particular OI-model, the user signals that new elements

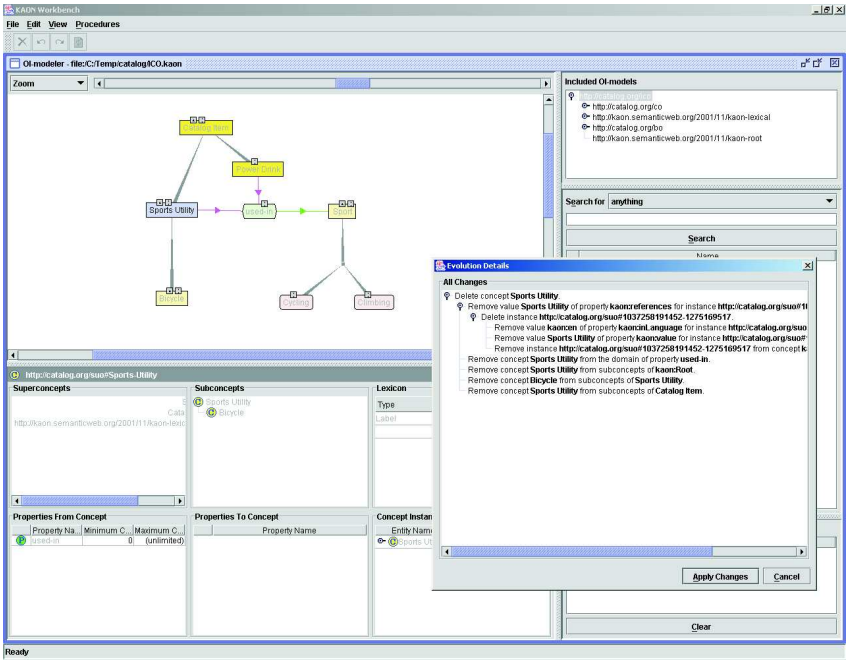

Fig. 11 OI-Modeler

should be created within this OI-model. The evolution subsystem takes care of maintaining of consistencies in all open models.

5 Related Work

In this section we discuss how our approach differs from other conceptual modeling approaches and tools available.

Classical Conceptual Modeling. Entity-relationship modeling is usually used for database design. During implementation are ER models transformed into a logical model – nowadays this is the relational model. Evolving and implementing such models is more complex than if only one paradigm were used, since the conceptual and logical perspective must be kept in synchrony. In our approach we isolate the mapping of the conceptual model to the logical one into a separate step, thus making the logical model hidden from the ontology user. The users of the ontology should use ontology in an ontology-natural way, while enjoying the benefits of relational database technology for information storage and management.

There have been proposals for using UML as an ontology modeling language (e.g. [8]). However, the reader may note that there are significant, but subtle, differences in standard object-oriented and ontology modeling. Classes of an object-oriented model are assigned responsibilities, which are encapsulated as methods. Often fictitious classes are introduced to encapsulate some responsibility. On the other hand, ontologies don't contain methods, so responsibility analysis is not important. In our conceptual modeling paradigm, however, these statements do not

hold. Membership of an object in a class is often interpreted as statement that some unary predicate is true for this object. If properties of an object change as time passes, then the object should be reclassified. Membership of an object in different classes at the same time has the notion that some different aspects are true for that object.

Ontology Modeling Languages. It has been argued by many (e.g. [27]) that the definition of RDFS is very confusing, because RDFS modeling primitives are used to define the RDFS language itself. Currently there is no specification for modularization of RDF models. The handling of lexical information is very messy – languages can be attached to RDF model using XML attributes. Further, only 1:m relationships between lexical entries with elements of the ontology are possible.

The ideas of object-oriented modeling paradigm on ontology modeling has resulted in creation of so called frame-based knowledge modeling languages, and F-logic [15] is one example. In F-logic, the knowledge is structured into frames (analogous to classes) that have different value slots (analogous to attributes). Frames may be instantiated, in which case slots are filled with values. F-logic introduces other advanced modeling constructs, such as expressing meta-statements about classes. Also it is possible to define Horn-logic rules for inferring new information which is not explicitly present. In comparison, our conceptual model approach introduces the notion of light-weight inferences, which are intended to be easily implementable with existing systems.

In [13] an ontology language SHOE has been proposed as basis for the Semantic Web. It distinguishes itself by an HTML syntax and a mechanism for embedding ontology fragments into HTML documents. Apart from that, the language provides primitives for modeling classes and relationships of any arity, as well as specifying Horn-type rules. Our approach distinguishes itself through the paradigm of light-weight inferences instead of providing a general rule facility, as well as the capabilities for meta-class modeling.

A large body of research has been devoted to a subclass of logic-based languages, called description logics (a good overview is given in [5]). Although description logics are founded on a well-research theory, as mentioned in [2], they have proven to be difficult to use due to the non-intuitive style of modeling. This is due to the mismatch between with predominant object-oriented way of thinking.

For example, a common knowledge management problem is to associate instances of the DOCUMENT concept with document's topics. In object oriented systems the only possible approach is to model all topics as members of the concept TOPIC, and to introduce subtopic transitive relation between topic instances. To an experienced object-oriented modeler, this solution will be intuitive. On the other hand, in description logic systems, since topics are arranged in a hierarchy, a common modeling solution is to arrange all topics in a concept hierarchy, to be able to reuse the subsumption semantics of description logics[6]. However, if topics are sets, what are the instances of this set? Most users think of topics as fixed entities, and not as (empty and abstract) sets. Relating some document d1 to a particular topic t1 is therefore awkward – d1 can't be related to a particular instance of t1, but to "some" instance of t1, like this: $\exists$has-topic.t1(d1).

[6] Thanks to Ian Horrocks for this discussion.

Another important ontology modeling language is OIL [12]. This language combines the intuitive notions of frames, clear semantics of description logics and the serialization syntax of RDF embedded within a layered architecture. Core OIL defines constructs that handles most of RDFS (without reification). Standard OIL defines constructs based on description logics. However, the syntax of the language hides the description logics background and presents a language with a more frame-based "feel". Instance OIL defines constructs for creation of instances. Despite its apparent frame-based flavor, OIL is in fact a description logic system, thus our comments about description logics apply to OIL as well. Although it supports ontology modularization, it doesn't have a consistent strategy for management of lexical information. Similar comments can be made about other ontology modeling languages founded in description logics, such as DAML+OIL [7] or OWL [9].

Managing Multiple Ontologies. Reusing ontologies in the Semantic Web context is hindered by the fact that the primary Semantic Web language – RDF(S) – doesn't provide means for including elements from other ontologies. Each RDF fragment can freely refer to any resource defined anywhere on the Web. This presents serious problems to tool implementors, since it is not possible to reason over the entire Web. Recognizing this shortcomings, many ontology languages, including but not limiting to OIL [12], DAML+OIL [7] and OWL [9], provide means for declarative inclusion of other models.

However, most tools simply use these declarations for reading several files at the beginning and then create an integrated model. OilEd – a tool for editing OIL and DAML+OIL ontologies developed as the University of Manchester – does exactly that: importing an ontology actually inserts a copy of the original ontology into the current ontology. As mentioned before in this paper, this has drawbacks related to ontology evolution. On the other hand, tools such as Ontolingua [11], offer support even for cyclical ontology inclusion, based on the formal definitions given in [10]. However, to the best of author's knowledge, these tools don't provide evolution of included ontologies. Protege-2000 [22] – a widely used tool for ontology editing developed at Stanford – provides the best support for ontology inclusion so far. In Protege it is possible to reuse definitions from a project by including an entire project. However, the implemented inclusion mechanism is too crude, as it doesn't allow extension of included entities. For example, it is not possible to re-classify or add a slot to a class in the including model. Further, only the outermost model may be changed, thus making the evolution of dependent ontologies impossible.

Evolution. In the last decade there has been very active research in the area of ontology engineering. The majority of researches in this area are focused on construction issues. However, coping with the changes and providing maintenance facilities require a different approach. We cannot say that there exist commonly agreed methodologies and guidelines for ontology evolution. Thus, there are very few approaches investigating the problems of changing ontologies.

In [14] it is pointed out that ontologies on the Web will need to evolve. Authors provide a new formal definition of ontologies for the use in dynamic, distributed environments. A web-based knowledge representation language SHOE is

presented. While supporting multiple version of ontologies and ontology reuse, the change propagation between distributed and dependent ontologies is not treated.

In contrast to the ontology evolution that allows access to all data only through the newest ontology, the ontology versioning allows access to data through different versions of the ontology. Thus, the ontology evolution can be treated as a part of the ontology versioning mechanism that is analyzed in [16]. Authors describe a system offering support for ontology versioning. In contrast to that approach, which detects changes by comparing ontologies, we track information about all performed changes, since the change detection is a complicated and a time-consuming process. Further, it is impossible to determine the cause and the consequences of a change, which is a crucial requirement for the consistency of the dependent ontologies.

Oliver et al. [24] discuss the kinds of changes that occur in medical ontologies and propose the CONCORDIA concept model to cope with these changes. The main aspects of CONCORDIA are that all concepts have a permanent unique identifier. Concepts are given a retired status instead of being physically deleted. Moreover, special links are maintained to track the retired parents and children of each concept. However, this approach is insufficient for managing a change on the Semantic Web, especially since there are no possibilities to control the whole process.

In [19] the author presents guiding principles for building consistent and principled ontologies in order to alleviate their creation, usage and maintenance in distributed environments. Authors analyze the requirements for the tool environments that enforces consistency. We have extended these operational guidelines by taking into account the usage of an ontology.

Other research communities also have influenced our work. The problem of schema evolution and schema versioning support has been extensively studied in relational and database papers ([1], [29]). In [23] authors discuss the differences that stem from different knowledge models and different usage paradigms.

Moreover, research in ontology evolution can also benefit from the many years of research in evolution of knowledge-based systems [26], [20]. The script-based knowledge evolution [31] that identifies typical sequences of changes to knowledge base and represents them in a form of scripts, is similar to our approach. In contrast to the knowledge-scripts that allow the tool to understand the consequences of each change, we go a step further by allowing the user to control how to complete the overall modification and by suggesting the changes that could improve the ontology.

The distributed ontology evolution problem is related to the research in distributed systems [25]. In [4] the authors develop techniques that combine push and pull synchronization in an intelligent and adaptive manner while offering good resiliency and scalability. We extend this approach by taking into account not only the coherency maintenance of the cached data but the maintenance of the dependent and replication consistency as well.

6 Conclusion

This article presented our integrated framework for managing multiple and distributed ontologies. We presented a conceptual modeling approach along with its

mathematical definition and formal semantics. We believe that such an approach is suitable as a basis for a scalable implementation, since it lends itself to implementation using existing database technologies by adding simple deductive capabilities. Our conceptual models can be reused through inclusion mechanisms. Further, reuse in a distributed setting is supported through replication. Another important part of our work is evolving ontologies. We isolated three separate cases: evolving single ontologies, evolving ontologies that have dependencies and evolving ontologies in a distributed setting.

For our future work we plan to focus on the following problems. First, we shall investigate how to better integrate and enrich our approach with similar approaches, notably description logics, but without sacrificing the performance. Second, the development of ontology mapping [17] and integration mechanisms with evolution support will be in focus of our future research. Finally, we plan to extend our approach for ontology evolution to not only to compute the necessary changes realizing user's requirements, but to also realize secondary goals, such as minimizing the total number of changes in the ontology. In such way we hope to provide an easier way for the user to specify and execute composite changes.

References

1. J. Banerjee, W. Kim, H.J. Kim, and H. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of the Annual Conference on Management of Data*, ACM SIGMOD, 1997.
2. S. Bechhofer, C. Goble, and I. Horrocks. DAML+OIL is not Enough. In *SWWS-1, Semantic Web working symposium*, Stanford (CA), July 29th-August 1st 2001.
3. C. Beeri and R. Ramakrishnan. On the power of magic. In *Proc. ACM SIGACTSIGMOD-SIGART Symposium on Principles of Database Systems*, pages 269–284, 1987.
4. M. Bhide, P. Deoasee, A. Katkar, A. Panchbudhe, and K. Ramamritham. Adaptive Push-Pull: Disseminating Dynamic Web Data. *IEEE Transaction on Computers*, June 2002.
5. A. Borgida. Description logics are not just for the FLIGHTLESS-BIRDS: A new look at the utility and foundations of description logics. Technical Report DCS-TR-295, Department of Computer Science, Rutgers University, 1992.
6. W. Chen, M. Kifer, and D. S. Warren. HiLog: A Foundation for Higher-Order Logic Programming. In *Journal of Logic Programming*, volume 15, pages 187–230, 1993.
7. D. Connolly, F. van Harmelen, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. DAML+OIL (March 2001) Reference Description, http://www.w3.org/TR/daml+oil-reference.
8. S. Cranefield and M. Purvis. UML as an ontology modelling language. In *Proc. of the Workshop on Intelligent Information Integration, 16th Int'l Joint Conf. on Artificial Intelligence (IJCAI-99)*, 1999.
9. M. Dean, D. Connolly, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL Web Ontology Language 1.0 Reference, W3C Working Draft 29 July 2002, http://www.w3.org/TR/owl-ref/.
10. A. Farquhar, R. Fikes, and J. Rice. Tools for assembling modular ontologies in Ontolingua.
11. A. Farquhar, R. Fikes, and J. Rice. The Ontolingua server: Tools for collaborative ontology construction. Technical report, Stanford KSL 96-26, September 1996.

12. D. Fensel, I. Horrocks, F. van Harmelen, S. Decker, M. Erdmann, and M. Klein. OIL in a Nutshell. In *Proc. 12th Int'l Conf. on Knowledge Engineering and Knowledge Management (EKAW-2000)*, Juan-les-Pins, France, October 2000.
13. J. Heflin. *J. Towards the Semantic Web: Knowledge Representation in a Dynamic, Distributed Environment*. PhD thesis, University of Maryland, College Park, 2001.
14. J. Heflin and J. A. Hendler. Dynamic Ontologies on the Web. In *Proc. 7th Nat'l Conf. on Artificial Intelligence AAAI-2000*, pages 443–449. AAAI/MIT Press, 2000.
15. M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the ACM*, 42:741–843, July 1995.
16. M. Klein, A. Kiryakov, D. Ognyanov, and D Fensel. Ontology Versioning and Change Detection on the Web. In *Proc. 13th European Conf. on Knowledge Engineering and Knowledge Management (EKAW-2002)*, Siguenza, Spain, October 2002.
17. A. Maedche, B. Motik, N. Silva, and R. Volz. MAFRA — An Ontology MApping FRAmework in the Context of the Semantic Web. In *Workshop on Ontology Transformation at ECAI - 2002*, Lyon, France, July 2002.
18. A. Maedche, B. Motik, L. Stojanovic, R. Studer, and R. Volz. Ontologies for Enterprise Knowledge Management. *To appear in: IEEE Intelligent Systems*, 2003.
19. D. McGuinness. Conceptual Modeling for Distributed Ontology Environments. In *Proceedings of The Eighth International Conference on Conceptual Structures Logical, Linguistic, and Computational Issues (ICCS 2000)*, Darmstadt, Germany, 2000.
20. T. Menzis. Knowledge Maintenance: The state of the Art. *The Knowledge Engineering Review*, page 10(2), 1998.
21. B. Meyer. *Object-oriented Software Construction (2nd Edition)*. Prentice Hall, 1997.
22. N. F. Noy, R. W. Fergerson, and M. A. Musen. The knowledge model of protege-2000: Combining interoperability and flexibility. In *Proc. 12th Int'l Conf. on Knowledge Engineering and Knowledge Management (EKAW-2000)*, Juan-les-Pins, France, October 2000.
23. N. F. Noy and M. Klein. Ontology Evolution: Not the Same as Schema Evolution. Technical Report SMI-2002-0926, Stanford, 2002.
24. D. E. Oliver, Y. Shahar, M. A. Musen, and E. H. Shortliffe. Representation of change in controlled medical terminologies. *AI in Medicine*, pages 53–8076, 1999.
25. M.T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall International, Inc., 1999.
26. M. Wrner P. Breche. How to remove a class in an ODBS. In *Proceedings of 2nd International Conference on Application Database*, Santa Clara, California, 1995.
27. J. Pan and I. Horrocks. Metamodeling architecture of web ontology languages. In *Proceedings of the Semantic Web Working Symposium*, pages 131–149, July 2001.
28. G. Pierre and M. van Steen. Dynamicaly Selecting Optimal Distribution Strategies on Web Documents. *IEEE Transaction on Computers*, June 2002.
29. J.F. Roddick. A Survey of Schema Versioning Issues for Database Systems. *Information and Software Technology*, pages 37(7):383–393, 1996.
30. L. Stojanovic, A. Maedche, B. Motik, and N. Stojanovic. User-driven Ontology Evolution Management. In *Proc. 13th European Conf. on Knowledge Engineering and Knowledge Management (EKAW-2002)*, Siguenza, Spain, October 2002.
31. M. Tallis and Y. Gil. Designing scripts to guide users in modifying knowledge-based systems. In *AAAI/IAAI*, pages 242–249, 1999.
32. C. A. Welty and D. A. Ferrucci. What's in an instance? Technical report, RPI Computer Science, 1994.