NeOn: Lifecycle Support for Networked Ontologies

Integrated Project (IST-2005-027595)

Priority: IST-2004-2.4.7 — "Semantic-based knowledge and content systems"

D1.3.1 Propagation Models and Strategies

Deliverable Co-ordinator: **Raul Palma**

Deliverable Co-ordinating Institution: **UPM**

Other Authors: **Peter Haase, Yimin Wang, Mathieu d'Aquin**

In this deliverable we present our approach for the management and propagation of ontology changes. The models and strategies introduced in this deliverable are also major components for case study task T7.4 that we used as a test case. In particular we analyse the workflow for collaborative ontology editing from task T7.4 and position the role of ontology propagation models and strategies in the collaborative workflow. Our approach addresses aspects related to the ontology change capturing and representation, the management of different ontology versions and the propagation of ontology changes in distributed environments. We also propose a workflow model for collaborative ontology editing and present methods for the ontology conflict resolution and ontology comparison. Finally, we introduce some initial implementations that apply these new developed models and strategies for networked ontology change management.

Document Identifier:	NEON/2008/D1.3.1/1.0	Date due:	November 30, 2007
Class Deliverable:	NEON EU-IST-2005-027595	Submission date:	January 14, 2008
Project start date	March 1, 2006	Version:	1.0
Project duration:	4 years	State:	Final
		Distribution:	Public

2006–2008 © Copyright lies with the respective authors and their institutions.

NeOn Consortium

This document is part of the NeOn research project funded by the IST Programme of the Commission of the European Communities by the grant number IST-2005-027595. The following partners are involved in the project:

Open University (OU) – Coordinator Knowledge Media Institute – KMi Berrill Building, Walton Hall Milton Keynes, MK7 6AA United Kingdom Contact person: Martin Dzbor, Enrico Motta E-mail address: {m.dzbor, e.motta}@open.ac.uk	**Universität Karlsruhe – TH (UKARL)** Institut für Angewandte Informatik und Formale Beschreibungsverfahren – AIFB Englerstrasse 11 D-76128 Karlsruhe, Germany Contact person: Peter Haase E-mail address: pha@aifb.uni-karlsruhe.de
Universidad Politécnica de Madrid (UPM) Campus de Montegancedo 28660 Boadilla del Monte Spain Contact person: Asunción Gómez Pérez E-mail address: asun@fi.ump.es	**Software AG (SAG)** Uhlandstrasse 12 64297 Darmstadt Germany Contact person: Walter Waterfeld E-mail address: walter.waterfeld@softwareag.com
Intelligent Software Components S.A. (ISOCO) Calle de Pedro de Valdivia 10 28006 Madrid Spain Contact person: Jesús Contreras E-mail address: jcontreras@isoco.com	**Institut 'Jožef Stefan' (JSI)** Jamova 39 SL–1000 Ljubljana Slovenia Contact person: Marko Grobelnik E-mail address: marko.grobelnik@ijs.si
Institut National de Recherche en Informatique et en Automatique (INRIA) ZIRST – 665 avenue de l'Europe Montbonnot Saint Martin 38334 Saint-Ismier, France Contact person: Jérôme Euzenat E-mail address: jerome.euzenat@inrialpes.fr	**University of Sheffield (USFD)** Dept. of Computer Science Regent Court 211 Portobello street S14DP Sheffield, United Kingdom Contact person: Hamish Cunningham E-mail address: hamish@dcs.shef.ac.uk
Universität Kolenz-Landau (UKO-LD) Universitätsstrasse 1 56070 Koblenz Germany Contact person: Steffen Staab E-mail address: staab@uni-koblenz.de	**Consiglio Nazionale delle Ricerche (CNR)** Institute of cognitive sciences and technologies Via S. Marino della Battaglia 44 – 00185 Roma-Lazio Italy Contact person: Aldo Gangemi E-mail address: aldo.gangemi@istc.cnr.it
Ontoprise GmbH. (ONTO) Amalienbadstr. 36 (Raumfabrik 29) 76227 Karlsruhe Germany Contact person: Jürgen Angele E-mail address: angele@ontoprise.de	**Food and Agriculture Organization of the United Nations (FAO)** Viale delle Terme di Caracalla 1 00100 Rome Italy Contact person: Marta Iglesias E-mail address: marta.iglesias@fao.org
Atos Origin S.A. (ATOS) Calle de Albarracín, 25 28037 Madrid Spain Contact person: Tomás Pariente Lobo E-mail address: tomas.parientelobo@atosorigin.com	**Laboratorios KIN, S.A. (KIN)** C/Ciudad de Granada, 123 08018 Barcelona Spain Contact person: Antonio López E-mail address: alopez@kin.es

Work package participants

The following partners have taken an active part in the work leading to the elaboration of this document, even if they might not have directly contributed writing parts of this document:

- Universidad Politécnica di Madrid (UPM)
- University of Karlsruhe (UKARL)
- Open University (OU)
- Ontoprise (ONTO)

Change Log

Version	Date	Amended by	Changes
0.1	25-05-2007	Yimin Wang	Creation
0.2	09-07-2007	Raul Palma	State of the Art
0.3	21-09-2007	Raul Palma	Workflow
0.4	31-10-2007	Raul Palma	Change Management
0.5	02-11-2007	Yimin Wang	System Architecture and Change capturing
0.6	05-11-2007	Raul Palma	Change Representation and Propagation
0.7	15-11-2007	Mathieu d'Aquin	Ontology Comparison
0.8	24-11-2007	Peter Haase	Conflict Resolution, ontology model, proof reading
0.9	26-11-2007	Raul Palma	format fixes
1.0	28-12-2007	Raul Palma, Peter Haase	Updates according to the quality assessor review

Executive Summary

In this deliverable we present models and strategies for propagating networked ontologies that are defined in NeOn Project Deliverable 1.1.1 and 1.1.2. The networked ontology propagation models and strategies introduced here are also major components for case study task T7.4. The recent networked ontology model calls for new models and strategies for networked ontology propagation, therefore it is necessary to summarize the existing work on in this field and develop new applications to fit the requirement from case study workpackages.
The networked ontology propagation models and strategies aim to help users in managing the changes of ontologies. Hence, we investigate several related tasks including the following aspects:

- ontology change capturing and representation
- ontology versioning
- workflow for collaborarative ontology editing
- ontology change propagation
- ontology conflict resolution
- ontology comparison

2006–2008 © Copyright lies with the respective authors and their institutions.

We first present a comprehensive state of the art and point out the current challenges in the existing approaches. Then, we analyse the workflow for collaborative ontology editing from task T7.4 and position the role of ontology propagation models and strategies in the collaborative workflow.

Based on the previous analysis we propose a layered approach for the modeling and formal representation of ontology changes. Next, we propose strategies for the management of different ontology verions. We introduce a formal model for the representation of the workflow used in the collaborative ontology editing. Then, we provide the methods and strategies for the management of propagation of ontology changes and finally methods for the comparison of ontologies and the management of conflicts. Afterwards, the implementation part provides some initial implementations that apply these new developed models and strategies for networked ontology change management.

In a nutshell, this deliverable reports the first step to provide comprehensive software prototypes as part of the NeOn Toolkit. We aim to continue our work in T1.3 by following the existing achievements and explore future advances in managing networked ontologies changes. The planned next step is to evaluate these models and strategies in T1.3.

Contents

1 Introduction 9
1.1 Motivation . . . 9
1.2 State-of-the-art . . . 10
1.2.1 Ontology versioning . . . 11
1.2.2 Change representation . . . 13
1.2.3 Collaborative Ontology Editing . . . 15
1.2.4 Change propagation . . . 16
1.2.5 Conflict resolution . . . 18
1.2.6 Ontology comparison . . . 19
1.3 Challenges . . . 20

2 Collaborative Workflow 21
2.1 Overview of the fisheries ontologies lifecycle . . . 21
2.2 Functional Requirements and Use Cases . . . 22
2.2.1 Use Cases Related to Editing and Workflow Activities . . . 22
2.2.2 Use Cases Related to Visualization . . . 23
2.2.3 Maintenance of changes . . . 24
2.3 Technical Requirements . . . 24

3 Ontology Change Management 26
3.1 Ontology Change Representation . . . 26
3.1.1 Overview of the Ontology Model . . . 26
3.1.2 Change Ontology . . . 28
3.1.3 Ontology Change Capturing . . . 30
3.2 Ontology Versioning . . . 31
3.3 Workflow Model for Collaborative Ontology Editing . . . 32
3.3.1 Workflow ontology . . . 34
3.3.2 Visualization of Ontologies in the Editorial Workflow . . . 37
3.4 Change Propagation . . . 38
3.4.1 Ontology Metadata . . . 39
3.4.2 Ontology Metadata Lifecycle . . . 40
3.4.3 Propagation of Ontology Changes to Ontology Metadata . . . 42
3.5 Conflict Resolution and Concurrent Ontology Editing . . . 45
3.6 Ontology Comparison . . . 46
3.6.1 Notations . . . 46
3.6.2 Semantic Comparison . . . 47

2006–2008 © Copyright lies with the respective authors and their institutions.

3.6.3 Syntactic Comparison . . . 47
3.6.4 Dealing with Anonymous Entities . . . 48
3.6.5 Vocabulary Comparison . . . 48
3.6.6 Overall Procedure . . . 49

4 Implementation **50**
4.1 Overall Software Architecture . . . 50
4.1.1 OWL Ontology Editor . . . 52
4.1.2 Change Capturing Component . . . 53
4.1.3 Versioning component . . . 54
4.1.4 Change Propagation Service . . . 55
4.1.5 Ontology Comparison . . . 55
4.1.6 Conflict Resolution . . . 55

5 Conclusions and Outlook **57**
5.1 Summary . . . 57
5.2 Future Work . . . 57

Bibliography **59**

List of Tables

3.1 Editorial workflow actions at the element level 34
3.2 Editorial workflow actions at the ontology level 35
3.3 Events, state transitions and validation actions 42

4.1 Mappings among requirements, methods and prototypes 52

2006–2008 © Copyright lies with the respective authors and their institutions.

List of Figures

3.1 OWL metamodel: ontologies 27
3.2 OWL metamodel: entities 27
3.3 Change ontology for OWL 1.1 30
3.4 The procedure of change capturing 31
3.5 Editorial workflow at the element level 33
3.6 Editorial workflow at the ontology level 33
3.7 Workflow ontology 36
3.8 Conceptual model for the Ontology Metadata Annotation 39
3.9 Ontology Metadata Annotation Lifecycle 40
3.10 Editorial Workflow & Metadata Lifecycle 43
3.11 Notification of Ontology Metadata Annotations changes 44

4.1 The overall architecture of workflow for ontology editing related components 51
4.2 The OWL Ontology Editor and its related components 53
4.3 The architecture of Change Capturing plug-in in the distributed networking scenario 54

Chapter 1

Introduction

In this chapter we provide a brief introduction by stating the motivation of this deliverable, followed by state-of-the-art study of ontology evolution approaches and technologies, and positioning the challenges in managing the networked ontology changes. In particular, the propagation models and strategies introduced here are based on the networked ontology model defined in Deliverable 1.1.1 [HRW$^+$06]. The networked ontology propagation models and strategies described in this deliverable are also major components for case study task T7.4 for fishery ontology life-cycle management. Therefore we use T7.4 as test case in this deliverable to show the applicability of the research work and direct link among workpackages. Nevertheless our approach is applicable to any other scenario with similar requirements.

1.1 Motivation

The widespread use and application of ontologies in many different areas during the last years, has increased the interest of researches in the construction of ontologies and the re-use of existing ones. This situation, however, demands also a bigger effort in the maintenance and management of ontologies. Experience demonstrates the fact that ontologies (as any other system component) are dynamic (i.e. they are evolving/changing throughout the time). Consider the ontology as the formal, explicit specification of a shared conceptualisation[SBF98], it is clear that ontologies might change whenever one of the elements of the definition change. For instance, domains are not static nor fixed, they might evolve e.g. because a non existing element becomes part of the domain or because some elements become obsolete. A similar situation occurs with the shared conceptualisation that might change during the time e.g. because the domain experts involved in the modeling of the ontology acquire additional knowledge about the domain. Finally, the specification might change e.g. because new ontology languages or new versions of the existing ones become available. In any case, the management of those changes involves the execution of many related tasks (i.e also known as ontology evolution process (see next section)). Even though we can find some recent works addressing either the whole process or some of the individual tasks, in general those works are considering partially the problem, and there are still many things that can be learned from other related fields. If we consider, for example, the task related to the representation of changes, we can find some ontologies proposed for capturing the changes, however none of them have become widely accepted and even if those ontologies are dependent on the underlying ontology model, they are not considering the real low level operations allowed in a specific ontology language. Even more important for the scope of this deliverable is the task considering the propagation of ontology changes to dependent artifacts (e.g. other ontologies, mappings, metadata, applications, etc.), as the issue has been hardly addressed at all.

The aforementioned situation becomes even more complex if we take into account the distributed nature of a network of ontologies where complex relations can exist between ontologies and consequently the necessity to the propagation of the changes to the distributed ontology dependent artifacts.

Finally, even if there exist many advanced tools for the development of ontologies (e.g. Protege, KAON2, WebODE, etc.), in general they are considering and supporting the single-user scenario where there is only

2006–2008 © Copyright lies with the respective authors and their institutions.

one user involved in the implementation of the ontology (and later modification). This is not the case in many organizations, like FAO, where the construction and maintenance of each ontology involves many people. Of course in order to give support for a collaborative ontology editing, we need the appropriate infrastructure.

1.2 State-of-the-art

The propagation models and strategies provide the foundation for one of the main activities involved in the ontology evolution process (i.e. the propagation of ontology changes). Hence, even if the ontology evolution in general is not the scope of this deliverable, we present the different approaches for the management of the evolution of ontologies. Although, the evolution of conceptual models such as schemas in databases or XML schemas, has been thoroughly investigated in the computer science field, the evolution of ontologies is still under continuous research. In the following we provide the most relevant works in this area.

Ontology evolution approaches Ontology Evolution is defined by [Sto04] as the timely adaptation of an ontology to the arisen changes and the consistent propagation of these changes to dependent artifacts. The author proposed a six phase process for the ontology evolution:

- The *change capturing* phase realizes the continual improvement of an ontology. It distinguishes two types of changes: top-down changes and bottom-up changes. The top down (deductive/explicit) changes are the result of the knowledge elicitation techniques that are used to acquire knowledge direct from human experts (domain experts or end-users). Bottom-up (inductive/implicit) changes match the machine learning techniques, which use the different methods to infer patterns from the sets of examples. The implicit changes are reflected in the behavior of the system and can be discovered only through the analysis of this behavior (e.g. Structure-driven Change Discovery, Data-driven Change Discovery, Usage-driven Change Discovery)

- The *change representation* phase is in charge of representing a request for a change formally and explicitly as one or more ontology changes. The authors proposed the representation of changes as instances of an evolution ontology (see subsection 1.2.2). This formal, explicit representation of ontology changes makes them machine-understandable, usable by other ontology evolution systems as well as exploitable for supplementary functionality of an ontology evolution system such as learnability.

- The *semantics of change* phase prevents inconsistencies by computing additional changes that guarantee the transition of the ontology into another consistent state. It enable the resolution of induced changes in a systematic manner, ensuring consistency of the whole ontology. In particular, the author focus on the structural inconsistencies that arise when the ontology model constraints are invalidated after a change request.

- In the *change propagation* phase all of the ontology dependent artifacts are updated. The related artifacts considered by the author include: distributed ontology instances, dependent ontologies and applications using the ontology that has to be changed (see subsection 1.2.4)

- The role of the *change implementation* phase is to inform the ontology engineer about all consequences of a change request, to apply all the (required and derived) changes to the ontology in a transactional manner and to keep track about performed changes.

- Finally, the *change validation* phase enables justification of performed changes and undoing them at user's request. This phase helps ontology engineers to find out whether they have built the right ontology, i.e. whether the changed ontology represents a piece of reality and the users' and/or application's requirements correctly (i.e. generation of the explanation).

Another approach for modelling the ontology evolution process is presented in [LM07]. It is based on the work proposed in [BR00] where they distinguish four generic activities in the process of making a change: requesting a change, planning the change, implementing the change, and verifying and validating the change.

- *Requesting a change* has to do with initiating the change process and includes activities related with the representation of changes, prioritisation of multiple changes and the discovery of changes.
- *Planning the change* has to do with understanding *why* the change needs to be made, and *where* the change needs to be made. Therefore, a crucial part of this activity has to do with change impact analysis where all the potential consequences (side effects) of a change are identified along with an estimation of what needs to be modified to accomplish a change. Finally an estimated cost of evolution is provided to allow the engineer to decide whether or not to implement the change.
- The *implementation of the change* involves the activities of change propagation, restructuring the ontology before the implementation of the change and inconsistency management.
- Finally, the *verification and validation* phase deals with the issues of building the right ontology and build it in a right way. Then, the activity of quality assurance will ensure that the developed ontology satisfies all desired qualities.

1.2.1 Ontology versioning

Although there is a clear distinction between schema evolution and schema versioning in the database community, the distinction is not so clear when applied to ontologies (i.e. ontology evolution and ontology versioning) and in fact they can easily lead to confusion. According to [Rod95], for databases, schema evolution is the ability to change a schema of a populated database without loss of data (i.e. providing access to both old and new data through the new schema) while schema versioning is the ability to access all the data (both old and new) through different version interfaces. For ontologies, in [Kle04], the author claims that there is no such distinction between ontology evolution and ontology versioning and that ideally developers should maintain not only the different versions of an ontology, but also some information on how the versions differ and whether or not they are compatible with one another. Hence, he combines ontology evolution and ontology versioning into a single concept defined as the ability to manage ontology changes and their effects by creating and maintaining different variants of the ontology.

In [OK02], the authors propose a model for tracking of changes, versioning, and meta-information for RDF(S) repositories. The approach is based on the fact that the RDF statement, which is the smallest directly manageable piece of knowledge, cannot be changed (i.e. it can only be added and removed). Hence, they deal with two basic types of updates in a repository, namely addition and removal of a statement. Each update change the repository into a new state which is defined as the set of statements that are explicitly asserted. Some of the states can be in turn considered as versions (i.e. depending on the user's or application's needs and desires) for which additional knowledge could be supported as a meta-information. The meta-information is supported for resources, statements, and versions. The authors only considers a small set of meta-information, but give the possibility to extend it on demand.

The issue of versioning was also addressed in [LM07], where the authors define it as a mechanism that allows users to keep track of all changes in a given system, and to undo changes by rolling back to any previous version. Furthermore, it can be used to keep track of the history of all changes made to the system. They also identify two variants of versioning: State-based versioning (i.e. at any given moment in time, the system under consideration is in a certain state, and any change made to the system will cause the system to go to a new state.) and change-based versioning where changes are treated as first-class entities (i.e. it stores information about the precise changes that were performed).

In [HH00] the authors present SHOE, a web-based knowledge representation language that supports multiple versions of ontologies. SHOE is described in the terms of a logic that separates data from ontologies and allows ontologies to provide different perspectives on the data. The paper presents the features of SHOE

2006–2008 © Copyright lies with the respective authors and their institutions.

that address ontology versioning, the effects of ontology revision on SHOE web pages, and methods for implementing ontology integration using SHOE's extension and version mechanisms.

[Kle02] introduces OntoView, a web-based change management system for ontologies. OntoView provides a transparent interface to different versions of ontologies, by maintaining not only the transformations between them, but also the conceptual relation between concepts in different versions. It uses several rules to find changes in ontologies and visualizes them (and some of their possible consequences) in the file representations. The user is able to specify the conceptual implication of the differences, which allows the interoperability of data that is described by the ontologies. Further, the authors describe the mechanism that were used to find and classify changes in RDFS / DAML ontologies. It also shows how users can specify the conceptual implication of changes to help interoperability.

In [KF01] the authors discuss the problem of ontology versioning based on work done in database schema versioning and program interface versioning. They define ontology versioning as the ability to handle changes in ontologies by creating and managing different variants of it. Further, they argue that in order to achieve this ability, we need a methodology with methods to distinguish and recognize versions, and with procedures for updates and changes in ontologies. This also implies keeping track of the relationships between versions. Based on the definition of ontologies by Gruber (1993), changes in ontologies can be caused by: (i) changes in the domain; (ii) changes in the shared conceptualization; (iii) changes in the specification. The authors also impose three requirements for a versioning framework: (i) ontology identification; (ii) change specification; (iii) transparent evolution.

One of the most interesting contributions of the previous paper is the discussion about the *identification of ontologies*. If an ontology is seen as a specification of a conceptualization, then every modification to that specification can be considered a new conceptualization of the domain. In that case, even syntactic changes and updates of natural language descriptions specify different concepts, which are per definition not equal (although they are fully compatible revisions). The authors take the following (practical) position: They assume that an ontology is represented in a file on the web. Every change that results in a different character representation of the ontology constitutes a revision. In case the logical definitions are not changed, it is the responsibility of the author of the revision to decide whether this revision is semantic change and thus forms an new conceptualization with its own identity, or just a change in the representation of the same conceptualization. Consequently there has to be a way to deal with the identification of ontologies on the web. This brings us into the very slippery debate on the meaning of URIs, URNs and resources[CEM01]. According to [BLFM98], a Uniform Resource Identifiers (URI's) is a *compact string of characters for identifying an abstract or physical resource*, where *a resource can be anything that has identity*. Hence, an ontology can harmlessly be regarded as a resource. Notice that URI's provide a general identification mechanisms, as opposed to Uniform Resource Locators (URL's), which are bound to the location of a resource. Accordingly, the authors propose a method to separate the identity of ontologies (ontology resources) completely from the identity of files (file resources) on the web that specify the ontology. Therefore, every revision is a new file resource and gets a new file identifier, but does not automatically get a new ontology identifier. The propose method however is not compliant with the the RDF Schema specification[BG04] which states that a new namespace URI should be declared whenever an RDF schema is changed. Yet this recommendation seems too strong and has already showed problems in practice (e.g. Dublin Core). Hence the proposed identification method is based on the following points:

- a distinction between three classes of resources: (i) files; (ii) ontologies; (iii) lines of backward compatible ontologies;
- a change in a file results in a new file identifier;
- the use of a URL for the file identification;
- only a change in the conceptualization results in a new ontology identifier;
- a new type of URI for ontology identification with a two level numbering scheme (i.e. mayor.minor[1]):

[1] In software versioning schemes, usually the major number is increased when there are significant jumps in functionality while

  - minor numbers for backward compatible modifications (an ontology-URI ending with a minor number identifies a specific ontology);
  - major numbers for incompatible changes (an ontology-URI ending with a major number identifies a line of backward compatible ontologies);

- individual concepts or relations, whose identifier only differs in minor number, are assumed to be equivalent;
- ontologies are referred to by an ontology URI with the according major revision number and the minimal extra commitment, i.e., the lowest necessary minor revision number.

1.2.2 Change representation

There exists some approaches for the formal and explicit representation of ontology changes. Much of the current work focuses on devising taxonomies of elementary change operators that are sound[2] and complete[3]. In [Sto04], the proposed so called evolution ontology to supports, alleviates and automates the ontology evolution process. It is about a meta-ontology that is used as a backbone for creating evolution logs. This ontology models what changes, why, when, by whom and how are performed in an ontology. However, the structure of the hierarchy of ontology changes is based on the KAON ontology model. The changes are classified as elementary or composite although the author introduces three levels of changes: elementary, composite and complex. An elementary change is an ontology change that modifies (adds or removes) only one entity of the ontology model, while a composite change represent a group of elementary changes applied together. Besides the hierarchy of ontology changes, the evolution ontology includes information supporting decision-making, such as cost, relevance, priority, textual description of the reason for a change etc. Furthermore, there are properties establishing the relations between changes (e.g. previousChange) along with their corresponding inverses. Finally, there are additional properties that depend on the change type (e.g. hasReferenceEntity). These properties are used to represent the peculiarities of a particular type of a change, such as its arguments. According to the author, the ŞhasReferenceEntityŤ property is not defined as a relation but as an attribute since it is also used to reference the entities from the domain ontology that do not exist anymore (i.e. its value is the unique identifier of the entity from the domain ontology that is changed).

In [Kle04], the author proposed an ontology for representing ontology changes for the OWL ontology model. In this ontology they model the relations between the most important concepts around ontology change. An actual change is specified as instance data that conforms to the ontology. Changes are classified as atomic or composite. According to the author, atomic operations are operations that cannot be subdivided into smaller operations and constitutes the top-level concept of the hierarchy of change operations for the OWL languages. Composite operations provide a mechanism for grouping a number of basic operations that together constitute a logical entity. The change concept is related to actual definitions of concepts via a ŞfromŤ and a ŞtoŤ property, which refer to the old and new version of a definition, respectively. They also define a concept that represent a set of change operations that has a source and target ontology. Finally, the ontology includes properties that specify arguments for a change operation as well as a property ŞeffectŤ to annotate the class of operations with the effect of the change.

Besides from the different underlying ontology model used by the previous approaches for the representation of ontology changes (i.e. KAON ontology model and OWL ontology model), there are other differences. The ontology proposed by [Kle04] is not minimal as it includes "modify" changes as a kind of atomic changes. On the other hand, [Sto04] approach only considers "add" and "remove" changes as elementary changes. Additionally, the former do not support reversibility (i.e. there is no information about the history of the changes) nor the necessary information for propagation (i.e. cause and consequences of change). Finally,

the minor number is incremented when only minor features or significant fixes have been added

[2]the manipulation operators should only generate valid ontologies

[3]the set should subsume every possible type of ontology access and manipulation

2006–2008 © Copyright lies with the respective authors and their institutions.

the former models change operations with references to the concepts or properties that they operate on, while the latter uses references to the domain entities through their identifiers, not through themselves.

Another related work is introduced in [NK03] where the authors identify a set of common complex changes combining the information of the structural differences between two versions of an ontology[NM02] with the hierarchical information. The complex changes identified include class moves and tree-level operations. However, those changes are not formally and explicitly represented and therefore they are not clearly defined.

Ontology changes can also be represented in other forms. According to [KN03] on the one end of the spectrum of representation forms, we may have very few details about changes from two versions of the ontology. On the other end of the spectrum, we may have a complete and detailed representation of changes from the two versions. In particular, the authors propose the following ways to represent change information for an ontology version:

- No explicit change information (i.e. there is only the old version of the ontology providing the basis for finding change information).
- A log of changes applied to the old version that result in the new version providing a record of the ontology-transition process. There are several detailed proposals for the information that logs should contain (e.g. In [Sto04], the author propose that the evolution log records an exact sequence of occurred changes as instances of an evolution ontology).
- A structural diff between versions that describes differences between them providing a declarative view of the ontology transition.
- A set of conceptual changes between versions providing an explicit specification of conceptual relations between concepts in the old version and corresponding concepts in the new version.
- A transformation set that describes a sufficient set of change operations for the transition from the old version to the new version providing an operational view of the changes.

Furthermore, the authors introduce an ontology of change operations for the OWL knowledge model similar to the one proposed by one of the authors in [Kle04]. This ontology however consists of two parts: The basis is an ontology of basic change operations (i.e. which is essentially the same as the one in [Kle04]) and there is an extension that defines complex change operations. Complex operations are defined as operations that are composed of multiple basic operations or that incorporate some additional knowledge about the change. However the extension is just a (non-hierarchical) list of some complex change operations that model specific variants of complex changes as subclasses of the root change class. The authors propose two methods for finding such complex changes: On the one hand, using a set of rules to generate a complex change from a set of basic changes. On the other hand, using heuristics to determine if a complex change occurred.

A similar ontology of changes for OWL-light knowledge model was previously presented by the same authors in [KFK+02]. At the highest level, the ontology makes a distinction between atomic changes and composite changes. Some of the composite changes are named, others are just unnamed aggregations of atomic changes. The named composite changes make it possible to specify an effect that is different than the combination of the effects of composite changes. Nevertheless, the effect and semantics of the composite change is not clearly specified in the ontology modelled in RDFS.

Based on the previous work, in [LAS06] the authors propose a so called Log Ontology to represent ontology changes. Similarly to the ontology of changes in [KFK+02], it models a hierarchy of the possible change operations during ontology evolution process but it differs also in some aspects: The Log Ontology only consider changes information on classes and properties, they did not consider the changes happened on the individuals. Also, they only consider elementary changes even though the ontology hierarchy include modifying and renaming operations. Finally, the perspective application makes the Log Ontology different with Klein's Ontology of Change. Log Ontology is used as a concept structure to organize the change information among the various versions of the same ontology in order to make the change process traceable.

Another approach for OWL DL ontologies was proposed in [HSV04] where the authors define atomic change operations of adding and removing axioms. Further as in previous approaches, composite ontology change operations can be expressed as a sequence of atomic ontology change operations. The work however does not elaborate further the proposed idea and does not refer to any concrete implementation.

The problem of representing changes has also been addressed in the past in other related fields. In [BKKK87] we can find a taxonomy and semantics of schema changes in object-oriented databases. They propose a classification of elementary changes including add, drop and change operations, based on the schema changes supported by the object-oriented database system called ORION. In [Ler00] the author introduces models for different levels of granularity in change operators for schema evolution: non-elementary or compound change operators. Similarly, another proposal for schema evolution in object databases is [PK97]. The authors provide a complete classification of modifications and propose three levels of modifications: primitive, composite and complex. The same three levels of modifications were reused by [Sto04] but tailored for KAON ontology model.

1.2.3 Collaborative Ontology Editing

The problem of collaborative ontology editing has been partially addressed in [NKKM04]. The authors claims that in an ontology-versioning environment, given two versions of an ontology, users must be able to: (1) examine the changes between versions visually; (2) understand the potential effects of changes on applications; and (3) *accept or reject changes*. They present the protege plugin PromptDiff[4] that allows users to accept and reject changes between versions at three levels of granularity: individual changes to property values and definitions, all changes in a definition of a class and all changes in class definitions and class-hierarchy positions in a subtree rooted at a particular concept. However, there are many issues related to the collaborative editing partially addressed (or not at all): the effects of accepting or rejecting changes (e.g. rejecting the addition of a class implies also reject the addition of its subclasses and their instances, automatically removing them from the new version) is implemented based on the expertise of the authors, is not possible to the editor to customize it. In addition, the authors are considering only the case when there are two different versions of an ontology without any information of the actual changes that led from one version to the other (i.e. they do not allow tracking changes directly during edition). In a collaborative ontology editing environment, it is not enough to know if a change was accepted or rejected, but also it is necessary to know the reasons of this decision. Finally, they are considering independent and local ontologies without taking into account the distributed nature and complex relations that can exist between ontologies and consequently the necessity to the propagation of the changes (i.e. accepted) to the ontology dependent artifacts.

A related work is presented in [dMLM06] (and further elaborated in [LM07]) where the authors propose a generic model for understanding the interorganisational ontology engineering process. Through this process the knowledge moves in an upward spiral starting at the individual level, moving up to the organisational level, and finally up to the interorganisational level. According to the authors an interorganizational ontology (IOO) consists of various, related sub-ontologies. The engineering process starts with the creation of an upper common ontology (UCO), which contains the conceptualisations and semantic constraints that are common to and accepted by a domain. Each participating organisation contextualises (through e.g., specialisation) this ontology into its own Organisational Ontology (OO), thus resulting in a local interpretation of the commonly accepted knowledge. In a so called Lower Common Ontology (LCO), a new proposal for the next version of the IOO is produced, selecting and merging relevant material from the UCO and various OOs. The part of the LCO that is accepted by the community then forms the legitimate UCO for the next version of the IOO.

In [Pin04] the authors introduce DILIGENT, an ontology engineering process for decentralized cases of knowledge sharing. It identifies several key roles (e.g. experts, with different and complementary skills) involved in collaboratively building the same ontology. DILIGENT comprises five main steps: (1) build, (2) local adaptation, (3) analysis, (4) revision, (5) local update.

[4] `http://protege.stanford.edu/plugins/prompt/prompt.html`

2006–2008 © Copyright lies with the respective authors and their institutions.

- *Build* The process starts by having domain experts, users, knowledge engineers and ontology engineers build an initial ontology.
- *Local adaptation* Once the core ontology is available, users work with it and, in particular, adapt it to their local needs. Logging local adaptations, the control board collects change requests to the shared ontology.
- *Analysis* The board analyzes the local ontologies and the requests and tries to identify similarities in users' ontologies. Since not all of the changes introduced or requested by the users will be introduced to the shared core ontology, a crucial activity of the board is deciding which changes are going to be introduced in the next version of the shared ontology.
- *Revise* The board should regularly revise the shared ontology, so that local ontologies do not diverge too far from the shared ontology.
- *Local update* Once a new version of the shared ontology is released, users can update their own local ontologies to better use the knowledge represented in the new version.

After the local update took place the iteration continues with local adaptation. During the next analysis step the board will review which changes were actually accepted by the users.

Another relevant work is presented in [TN07], where the authors introduce an extension of the existing Protégé[5] system that supports collaborative ontology editing (i.e. Collaborative Protégé). Besides the common ontology editing operations supported in Protégé, the extension enables the annotation of both ontology components and ontology changes. This functionality allows the users to comment and discuss about the content and changes of the ontology that they develop in common. Additionally, the Collaborative Protégé supports the searching and filtering of user annotations based on based on simple or complex criteria. Finally, the authors propose two types of voting mechanisms that can be used for voting change proposals (i.e. a *5-star* voting or a *Agree/Disagree* type of voting). However the current prototype does not support a workflow for the voting mechanism.

1.2.4 Change propagation

The issue of propagating changes has been addressed in the past in related areas (e.g. databases, distributed systems), however for the ontology domain, there are only a few works regarding the propagation of ontology changes and in most of this work the issue is just partially addressed.

Building new ontologies reusing existing ones instead of building them from scratch every time has many benefits (e.g. lowers the time and cost of developing new ontologies, avoids duplicate efforts, ensure interoperability, etc.). However, reusing ontologies, raise new challenges. As ontologies are rarely static, every time an ontology is changed locally it has to be taken into account the dependencies between ontologies (i.e. propagate those changes to dependent ontologies and other related artifacts). Furthermore, those dependencies become even more complicated in our envisioned scenario of NeOn, where ontologies are spread across many different nodes (servers).

The problem of propagating ontology changes is addressed by [Sto04] by first introducing two basic building blocks for realising reuse: Ontology inclusion and ontology replication. The former allows reusing ontologies available within the same node while the latter enables inclusion in the case when ontologies are distributed on different ontology servers (nodes). The ontology inclusion proposed by the author is based on the KAON ontology language and is limited to including entire models rather than including subsets. Also, when a model is reused, information can only be added, and not retracted. Finally the author do not deal with semantic inconsistencies between included ontologies. In a more realistic scenario where ontologies are spread across many different nodes (servers), the author propose replicate distributed ontologies locally and to include them in other ontologies. Replicated ontologies should never be modified directly. Instead, the ontology should

[5] `http://protege.stanford.edu`

be modified at the source and changes should be propagated to replicas using the distributed evolution process proposed by the author. Based on the previous notions, the propagation of changes is handled at two different levels: propagation of changes for multiple dependent ontologies within a single node (so called dependent ontology evolution) and the propagation of changes for multiples ontologies at multiples nodes (so called distributed ontology evolution). For the dependent ontology evolution problem, the author propose the synchronisation between dependent ontologies using a push-based approach (i.e. changes from the changed ontology are propagated to dependent ontologies as they happen). Even more, since permanent consistency of ontologies within one node is necessary, changes are propagated immediately, as they occur. For the distributed ontology evolution, the author introduces two problems to solve: the propagation of changes from an original to its replica (called the replication ontology consistency) and the propagation of changes from a replica to an ontology that includes it (called the dependent ontology consistency). The proposed solution employs a hybrid synchronisation strategy where the first task is addressed using a pull approach for synchronising originals and replicas and for the second task a push approach is used for maintaining consistency of ontologies within one node. Therefore, the distributed ontology evolution process used to guarantee the consistency of distributed ontologies is derived by using the pull synchronisation strategy between the original and its replicas and by applying the dependent evolution process described above to deltas (i.e. changes that have been applied to the original since the last synchronisation of the replica).

In [Oli00] the authors present an approach for the management of change that comprises the ability to make copied (and possibly changed) versions of controlled vocabularies (e.g.ontologies) up to date with a remotely changed controlled vocabulary. For a local site, there is a tradeoff between having autonomy over a local vocabulary and conforming to a shared vocabulary to obtain the benefits of interoperation. If the local site is motivated to conform, then the burden lies with the local site to manage its own changes and to incorporate the changes of the shared version at periodic intervals. The author call this process *synchronization* and is defined as the application of shared-vocabulary changes to the modified local version of a shared vocabulary to reach a target state. Since the goal of this process is to Şupdate the local vocabulary to obtain the benefits of shared-vocabulary updates, while maintaining local changes that serve local needsŤ, it can be considered as a distributed ontology evolution process. However, the underlying model is very simple, which results in few types of changes.

One related problem to the propagation of changes is the maintenance of coherency of related objects (i.e. changes in a dependent object have to be propagated to depending objects to keep them coherent). For example, an important issue in the dissemination of time-varying web data such as sports scores and stock prices is the maintenance of temporal coherency. In [DKP+01] there is an evaluation of push and pull algorithms to maintain such coherency. In the case of servers adhering to the HTTP protocol, clients need to frequently pull the data based on the dynamics of the data and a user's coherency requirements. In contrast, servers that possess push capability maintain state information pertaining to clients and push only those changes that are of interest to a user. These two canonical techniques have complementary properties with respect to the level of temporal coherency maintained, communication overheads, state space overheads, and loss of coherency due to (server) failures. The paper show how to combine push- and pull-based techniques to achieve the best features of both approaches. The combined technique tailors the dissemination of data from servers to clients based on (i) the capabilities and load at servers and proxies, and (ii) clients' coherency requirements.

The problem of change propagation has been thoroughly investigated for schema evolution ([PÖ97], [NR89], [PS87], [LH90], [RR97], [BKKK87], etc.). Although in data schema evolution the principal dependent artifacts are the instances representing the database population, the work can be reconsidered for the ontology domain (e.g. for updating ontology instances). In general, either objects are explicitly coerced to coincide with the new definition of the schema (i.e. update the affected objects, changing their representation as dictated by the new schema) or a new version of the schema must be created leaving the old version intact. Proposed approaches include screening (also known as deferred propagation), conversion (also known as immediate propagation), and filtering as techniques to define when and how coercion takes place. In screening, schema changes generate a conversion program that is independently capable of converting objects into the new

2006–2008 © Copyright lies with the respective authors and their institutions.

representation. The coercion is not immediate, but rather is delayed until an instance of the modified schema is accessed. In conversion, each schema change initiates an immediate coercion of all objects affected by the change. This approach causes processing delays during the modification of schema, but delays are not incurred during object access. Another solution is to introduce a new version of the schema with every modification and supplement each schema version with additional definitions that handles the semantic differences between versions. These additional definitions are known as filters and the technique is called filtering. In the filtering approach, changes are never propagated to the instances. Instead, objects become instances of particular versions of the schema. There are also hybrid approaches that combines two or more of the above methods.

Another relevant work is presented in [BR00]. The paper presents a model of change propagation during software maintenance and evolution. Change propagation is modelled as a sequence of snapshots, where each snapshot represents one particular moment in the process, with some software dependencies being consistent and others being inconsistent. A snapshot is changed into the next one by a change in one software entity and the dependencies related to it. The formalism for this process is based on graph rewriting. The paper discusses two basic processes of change propagation: change-and-fix, and top-down propagation. The change-and-fix process is the most common process of software maintenance. It allows changes to propagate in all directions and the first change can begin anywhere. However many of its properties are unclear. For example, it is conceivable that the entities will be visited several times or even that there may be an infinite process, where the change propagates in a circle and repeatedly revisits the previously changed entities. The top-down approach is a more predictable process, where changes propagate only from top-down and the first change can begin just in the topmost class.

Finally, we would like to include a brief overview of another related technology that can be useful for our issue of propagating changes. RSS[6] (which, in its latest format, stands for "Really Simple Syndication") is a family of web feed formats used to publish frequently updated content such as blog entries, news headlines or podcasts. An RSS document (also known as a "feed", "web feed" or "channel") contains either a summary of content from an associated web site or the full text. RSS content can be read using software called a "feed reader" or an "aggregator." The user subscribes to a feed by entering the feed's link into the reader or by clicking an RSS icon in a browser that initiates the subscription process. The reader checks the user's subscribed feeds regularly for new content, downloading any updates that it finds. Even though the kinds of content delivered by a web feed are typically HTML (webpage content) or links to webpages and other kinds of digital media, they could be adapted (at least the idea) to our domain (i.e. updates to ontologies).

1.2.5 Conflict resolution

The issue of resolving conflicts has been addressed in many different domains. In our context (i.e. ontologies) the authors in [LM07] identify three main characteristics of conflict: interaction, interdependence, and incompatible goals. Goals should be understood as meaning and incompatible meaning refers to the divergent ontological elements caused by alternative perspectives (i.e. known as cognitive conflict[D05]). One activity that illustrates the need for conflict resolution is ontology merging that refers to the activity of integrating changes that have been made in parallel to the same or related artifacts, in order to come to a new consistent system that accommodates these parallel changes. This is specially true in a collaborative setting where different persons can make changes simultaneously and often without even being aware of each other's changes. Furthermore, the authors identify different inter- and intra-ontological context dependency types: (i) articulation; (ii) application; (iii) specialisation; (iv) revision; and (v) axiomatisation. Those dependencies constrains the possible combinations of operators that can be applied to an ontology. For each of those requirements, they distinguish different types of conflicts for which support needs to be provided (e.g. ambiguous term, differentia undefined, etc.).

[6] http://cyber.law.harvard.edu/rss/rss.html

1.2.6 Ontology comparison

There are in the literature many approaches on how to compare two different version of ontologies in order to find the differences. One of such approaches is presented in [NM02] where the authors propose PROMPTDiff, an algorithm to find differences between two versions of a particular ontology. It is based on the comparison of versions of software code which entails a comparison of text files. Code is a set of text documents and the result of comparing the documents (the process is called a diff) is a list of lines that differ in the two versions. However text-file comparison is largely useless in comparing versions of ontologies (i.e. two ontologies can be exactly the same conceptually but have very different text representations. For example, their storage syntax may be different. The order in which definitions are introduced in the text file may be different.). PROMPTDiff compares the structure of ontology versions and not their text serialization. The algorithm consists of two parts: (1) an extensible set of heuristic matchers and (2) a fixed-point algorithm to combine the results of the matchers to produce a structural diff between two versions. Each matcher employs a small number of structural properties of the ontologies to produce matches. The fixed-point step invokes the matchers repeatedly, feeding the results of one matcher into the others, until they produce no more changes in the diff.

In [Kle02] the authors also present an algorithm used to compare ontologies in the system ontoview (a system that provides support for the versioning of online ontologies). OntoView, compares version of ontologies at a structural level, showing which definitions of ontological concepts or properties are changed. The algorithm uses the fact that the RDF data model underlies a number of popular ontology languages, including RDF Schema and DAML+OIL. The RDF data model basically consists of triples of the form <subject, predicate, object>, which can be linked by using the object of one triple as the subject of another. There are several syntaxes available for RDF statement, but they all boil down to the same data model. A set of related RDF statements can be represented as a graph with nodes and edges. Based on that fact, the algorithm split the document at the first level of the XML document. This groups the statements by their intended ŞdefinitionŤ. The definitions are then parsed into RDF triples, which results in a set of small graphs. Each of these graphs represent a specific definition of a concept or a property, and each graph can be identified with the identifier of the concept or the property that it represents. Then, they locate for each graph in the new version the corresponding graph in the previous version of the ontology. Those sets of graphs are then checked according to a number of rules. Those rules specify the ŞrequiredŤ changes in the triples set (i.e., the graph) for a specific type of change. The rules are specific for a particular RDF-based ontology language (e.g. DAML+OIL), because they encode the interpretation of the semantics of the language for which they are intended.

Another relevant work presented in [Völ06] where that introduces SemVersion, a generic, extendable multi-language ontology versioning system. According to authors, a semantic versioning system needs the ability to compute a diff, which tells the user what (conceptually) has changed between two versions. They distinguish three levels of diffs:

- Set-based Diff which is simply the set-theoretic difference of two RDF triple sets. Such diffs can be computed by simple set arithmetics for triple sets that contain only URIs and literals, however this level may result in large diffs.

- Structural Diff which takes blank node semantics into account. Without the presence of blank nodes, the set-based diff is the same as the structural diff. With blank nodes, the set-based diff considers all blank nodes to be different an reports all statements involving blank nodes both as added and as removed.

- Semantic Diff which respects the ontology language semantics. Formally, to calculate the semantic diff dl under the semantic of an ontology language l, a system has to know the semantics of that specific language l. Briefly, a way to compute a semantic diff is to materialize the complete entailment (transitive closure) and then perform a structural diff.

2006–2008 © Copyright lies with the respective authors and their institutions.

1.3 Challenges

In the previous section we presented a comprehensive state of the art survey on all the activities related to the propagation of ontology changes. As we noted, a large number of those activities (e.g. change propagation, change representation, versioning approaches, etc.) have been thoroughly investigated in many other computer science fields, however for our domain (i.e. ontologies), they are still under active research. In the rest of this section we will briefly discuss some of the main challenges for the core topics of this deliverable.

We presented in section 1.2 some approaches for the representation of ontology changes. In general those approaches classify ontology changes as elementary (atomic) or composite and some approaches also consider a complex type of change. Even though the proposed elementary (atomic) changes are introduced as operations that cannot be subdivided into smaller operations, they are in all cases, considering the change at the entity level (i.e. concepts, properties, individuals), and in some cases they are not even minimal (e.g. include modify operations). Furthermore, existing approaches are dependent on the underlying ontology model (i.e. the are based on proprietary models (e.g. KAON) or for specific languages (e.g. OWL) and consequently they have different set of elementary (atomic) changes. Hence, one key challenge on this area, would be to provide a language-independent approach with an additional lower level for the type of ontology change that will be the actual "minimal" operation that can be performed in a particular ontology model. From this generic ontology, one can have many specializations (e.g. for the different ontology languages). For instance in NeOn where we are considering OWL ontologies, a specialization of the generic ontology should specify that the lower level includes adding and removing OWL axioms.

Regarding the propagation of ontology changes still much can be learned from other related fields. Although the issue has been addressed in fields like databases, the state of the art shows that only a few works exist for the ontology field and in most of that work the issue is just partially addressed. For instance, the propagation of ontology changes to the related ontology metadata (and the corresponding metadata synchronization) has not even been considered. Consequently, a key challenge will be to evaluate the technologies that have been successfully applied in other domains and propose methods and strategies for the propagation of ontology changes.

Similarly, section 1.2.3 shows that many issues related to the collaborative aspects for the ontology editing have been just partially addressed (or not at all). For instance, in the foreseen scenario for networked ontologies, we should think about the distributed nature and complex relations that can exist between ontologies and consequently the necessity to the propagation of the changes (i.e. accepted) to the ontology dependent artifacts. Furthermore, in a more complex setting like the FAO Fisheries ontologies lifecycle (see section 2), where different kinds of ontology editors (possible concurrently) consult, and if authorized, validate and/or modify ontologies in a controlled and coherent manner, we will need the appropriate means to support the whole process, including the management and identification of different ontology versions, the formal representation of changes, conflict resolution, ontology comparison, etc.

Chapter 2

Collaborative Workflow

In this chapter we describe the requirements that we need to address in order to support a collaborative workflow. According to [GLP+07], a collaborative workflow in our context (i.e. ontologies) is a a special case of epistemic workflow characterized by the ultimate goal of designing networked ontologies and by specific relations among designers, ontology elements, and collaborative tasks. It is one of the main components for the ontology design.

Furthermore, the collaborative workflow is the central procedure for ontology change management. A typical workflow starts with proposals for ontology changes. These proposals are discussed by multiple users in a collaborative way. If a change is made, it has to be approved. After that, the change will be considered definitive and permanently added to the structure.

For the analysis of the collaborative workflow we are using as test case the editorial workflow of the fisheries ontologies lifecycle from task 7.4[MGKS+07].

2.1 Overview of the fisheries ontologies lifecycle

One of the goals of the FAO use case partner is that fisheries ontologies produced within WP7 will underpin the Fisheries Stock Depletion Assessment System (FSDAS).

However, for such a dynamic domain like fisheries that is continuously evolving, we will need to provide the appropriate support for a successful implementation and service delivery of the FSDAS. In particular, it will be crucial to support *ontology editing* and *maintenance* activities in order to incorporate and continuously reflect *changes* in the domain in the related ontologies.

The full lifecycle of the fisheries ontologies is introduced in [MGKS+07]. It consists of six major steps: First, ontology engineers organize and structure the domain information (i.e. from the Fisheries FIGIS databases, Fisheries fact sheets and other information system and documents) into meaningful models at the knowledge level (*conceptualize*). In the next step, ontology engineers perform the knowledge acquisition activities with various manual or (semi)automatic methods various methods to transform unstructured, semi-structured and/or structured data sources into ontology instances (*population*). The third step is the iteration of conceptualization and population processes until getting a populated ontology that satisfies all requirements and it is considered stable. Once achieved, on step four, the ontology will enter into the test and maintenance environment, implemented through the editorial workflow in step five. The editorial workflow will allow Ontology editors to consult, validate and modify the ontology keeping track of all changes in a controlled and coherent manner. Finally, once ontology editors in charge of validation consider the ontology final, they are authorized to release it on the Internet and make it available to end users and systems.

2006–2008 © Copyright lies with the respective authors and their institutions.

2.2 Functional Requirements and Use Cases

In this section we analyse the functional requirements of the editorial workflow, i.e. the specific functionality that show how the editorial workflow is going to be satisfied, including the specification of the workflow behavior. In this section we describe the particular use cases from the FAO that we will address in the rest of this deliverable.

Lifecycle Requirements The Fisheries editorial workflow should implement the necessary mechanisms to allow Ontology editors to:

- *consult* ontologies
- *modify* ontologies when they are authorized
- *validate* ontologies only if they are authorized
- *publish* ontologies on the internet only after ensuring they are fully validated.

Furthermore, the process should ensure that the aforementioned activities are carried out in a controlled and coherent manner. Hence, the editorial workflow is responsible for the coordination of who (depending on the user *role*) can do what (i.e. what kind of *actions*) and when (depending on the *status* with the ontology element (e.g. classes, properties and individuals) and the role of the user).

The users of the editorial workflow are ontology editors that are in charge of the everyday editing and maintenance work of the networked multilingual ontologies. Depending on the users permissions, they can be in charge of developing specific fragments of ontologies, revising work done by others, and developing multilingual versions of ontologies. They know about the ontologies domain, but usually know little or nothing about ontology software or design issues. Consequently, an ontology editor can be assigned one of the following roles:

- *Subject expert* are the editors inserting or modifying ontology content.
- *Validators* revise, approve or reject changes made by subject experts, and they are the only ones who can copy changes into the production environment for external availability.
- *Viewers* are users authorized to enter in the system and consult approved information about ontologies but they cannot edit the ontologies.

To control when are the ontology editors allowed to work with an ontology element, in addition to the user roles, every ontology element is required to have a status. Ontology editors can change the status depending on their role. For instance, when a subject expert insert a new element to the ontology, the new element will be assigned a draft state, indicating that some validator will have to validate it before it becomes part of the ontology.

Finally, an additional requirement is the set of possible actions that specify what are the particular operations that ontology editors are allowed to perform to the ontology elements depending on their user role.

2.2.1 Use Cases Related to Editing and Workflow Activities

According to the use cases described in [MGKS^{+}07], the UC-15 is covering the operations required to support the editorial workflow. This includes the operations the ontology editors are allowed to perform depending on their roles and the status of the ontology elements (i.e. classes, properties and instances).

UC-15.1: Edit Ontology Element Ontology editors (subject experts and validators) are allowed to edit ontology elements, depending on the status of the elements and their user rights and roles.

UC-15.1.1: Insert an ontology element Subject experts are allowed to insert new elements. When subject experts insert a new element, the new element is assigned the Draft status and triggers to start the editorial workflow.

UC-15.1.2: Update an ontology element Subject experts and validators are allowed to update ontology elements, depending on their role and the status of the element: A subject expert can only update elements in Draft or Approved status. In both cases the status of the element is automatically reset by the system to "Draft" status, and the element will need to pass through the whole workflow again to be released. A validator can update elements in "To be Approved" and "Approved" status; after the update the element keeps the status it had before the update.

UC-15.2: Delete an ontology element Ontology editors are able to propose for deletion an ontology element that is in the "Approved" status. In any case this is not a definitive action, and the element is automatically assigned the "To be deleted" status. Only validators are able to definitely destroy an element in the "To be deleted" status. Additionally, subject experts are able to delete ontology elements in "Draft" status but in this case the element is destroyed immediately (i.e. it does not need to be approved by a validator).

UC-15.3: Change status of element While inserting, updating and deleting elements, the status of those are automatically changed, when required, by the system. There are other cases where a specific action by ontology editors is required to move an element from one status to another and make the editorial workflow to function.

UC-15.3.1: Send to "To be Approved" status Subject experts are allowed to select an element in Draft status and sent it to the validator for approval, by changing the status of the element from "Draft" to "To be approved".

UC-15.3.2: Send to the "Approved" status Validators are allowed to approve elements, and thus to move them from the "To be Approved" status to the "Approved" status.

UC-15.3.3: Reject to "Draft" status Validators are allowed to reject element proposed to them to review in the "To be approved" status and send them back to the "Draft" status for the Subject Expert to check, update or complete.

UC-15.3.4: Reject to "To be Approved" status Validators are allowed to move back to the "To be approved" status an element already approved, if they consider it necessary.

UC-15.3.5: Reject the "To be deleted" proposal for an element Validators are allowed to reject a proposal for an element's deletion that are on the "To be deleted" status. In this case the element goes back to the "Approved" status.

UC-15.3.6: Accept the "To be deleted" proposal for an element Validators are able to accept the deletion and definitely destroy an element in the "To be deleted" status.

UC-15.4: Publish Ontology Validators are allowed to copy an ontology where all its elements are in the "Approved" status, from the test and validation environment (editorial workflow in the Intranet) to the production environment (Internet). By doing so, the system automatically assigned the right version to the published ontology. From V1 for the first time a particular ontology goes live, to Vn+1 for the N upgrade of the ontology.

2.2.2 Use Cases Related to Visualization

UC-9.2: View changes history Ontology editors are able to view the logs about changes to ontology elements (cf. UC-9.1), including the creation/editing date and history notes (to track changes to a concept).

View based on status and user role Hence, the interface should be able to provide different data views based on the user role. For instance, a validator should be able to view all the changes proposed by subject

2006–2008 © Copyright lies with the respective authors and their institutions.

experts that are still not validated or accepted in addition to the accepted ontology elements. In any case it should be clearly visible the difference between the different states of the elements in the ontology.

UC-9.3: View use statistics Ontology editors can view important information about an ontology. At least: provenance; previous editors; frequency of changes; and the fragment/domain of the ontology changed most rapidly.

UC-9.4: View ontology statistics Authorized users can view statistics of the ontology being edited. At least: depth of the class hierarchy; number of child nodes; number of relationships and properties; number of concepts per branch.

2.2.3 Maintenance of changes

The UC-9 is mostly related to the necessary metadata that has to be maintained to support the editorial workflow (i.e. to keep track of the ontology changes). This includes, the logging of those changes, and the corresponding visualization.

UC-9.1: Capture ontology changes The system logs changes in all ontology elements. That is, the date of creation/editing, the editor that made the change, etc. All actions are logged with the following minimum fields:

- description
- operation executed
- timestamp
- URI
- user

Change Propagation and Notification After new changes are send to the ontology, all editors involved in this workflow process are informed by a warning message when they log into the system. When modules are used to share authorial/editorial duty, each author (or the designated coordinator) should be able to view changes made by other authors, even without editing permission.

2.3 Technical Requirements

In this section we introduce the requirements for the technology solution that will support the aforementioned functional requirements for the editorial workflow.

Representation of changes A core requirement is the formal and explicit representation of the changes that ontology editors are able to perform to the FAO ontologies. To ensure the accessibility and interoperability with other components, the representation of changes should be formalized as an ontology (i.e. change ontology). Additionally, in order to support the different status that each ontology element can have during the editorial workflow (e.g. draft, approved, etc.), the change ontology should be able to capture the changes at the element level. Of course, to facilitate the editing, the change ontology might also consider changes at a higher level of abstraction (e.g. move a tree of concepts). As we mentioned earlier, in our scenario we are mainly considering OWL ontologies and hence an additional desired requirement for the change ontology would be to model the corresponding set of actual "atomic" operations that can be performed over OWL

ontologies (i.e. add/remove axioms) to provide an efficient link between what the user see (i.e. elements) and what the system manage internally (i.e. axioms). Having a track of the changes at different levels, will support the visualization of the ontology elements with a clear distinction of the state of each element of the ontology.

Versioning An additional requirement is the management of the different ontology versions. The first modification to an approved/published ontology automatically changes the current version. This modified version of the ontology will either become a new version (i.e. with a different version information) or if specified by the editor remain the same version (i.e. with the same version information). Whenever a new version is created, either is approved or not later on, we will need a way to uniquely identify the different versions of the ontology. Once more, the change ontology provides the necessary means to support not only the tracking of the changes but also the information that identifies the original version of the ontology and the current version of the ontology after applying the changes. This is not a trivial issue: even though ontologies are in general identified by an URI, in practice the URI is not enough to identify a particular version of an ontology (i.e. many different versions of the same ontology have the same URI). As a consequence, the management of ontology versions requires a clear definition of the ontology identification.

Conflict Resolution The editorial workflow should give support to many different ontology editors (e.g. subject experts, validators, etc.). Hence, an important issue that has to be addressed to ensure the consistency of the ontology is the resolution of conflicts whenever two or more editors submit changes to the same element concurrently. Even though, the expected frequency of concurrent modifications is low, the system should be able to ensure the consistency of the ontologies in every moment.

Representation of the workflow Yet another requirement is the formal representation of the workflow process. As it has been described above, the editorial workflow is concerned about user roles, status of the ontology elements and actions that can be performed to the ontology elements. Similarly to the representation of changes, to ensure the accessibility and interoperability with other components, the representation of the workflow process should be formalized as an ontology (i.e. workflow ontology). Having both models (i.e. ontology changes and workflow process) formalized as ontologies will facilitate the representation of the tight relationship that exists between both of them. For instance, consider a user with role "subject expert" that "inserts" a new ontology "concept" to the ontology. That "concept" will receive automatically the "draft" state. All the information related to the process of inserting a new ontology element will be captured by the workflow ontology, while the information related to the particular element inserted, along with the information about the ontology before and after the change is captured by the change ontology.

2006–2008 © Copyright lies with the respective authors and their institutions.

Chapter 3

Ontology Change Management

In this chapter, we introduce our approach for managing ontology changes. Our approach features a layered model formally represented as an ontology for describing the information about changes of ontologies. Based on this ontology for ontology changes (i.e. change ontology), we introduce integrated components to support the ontology change management. In this first version of the ontology change management infrastructure, we propose methods for the following aspects:

- A layered approach for the representation of ontology changes and the ontology change capturing component that captures changes that have applied to ontologies
- Ontology versioning management that supports the version control of ontologies
- Workflow model for the management of the collaborative ontology editing
- Models and strategies for the propagation of ontology changes
- Conflict resolution management to help user in handling the conflicts of user-actions in editorial workflow, for example, actions from two different users are controversial
- Ontology difference checking aims to identify whether two ontologies are different and what the difference are to help user in selecting a correct ontology

3.1 Ontology Change Representation

3.1.1 Overview of the Ontology Model

An important design decision is the selection of the ontology language to be supported in the change representation. In our work, we rely on the Networked Ontology Model as described in [HBP^{+}07]. The Networked Ontology Model is defined using a metamodeling approach based on MOF (Metaobject Facility). The metamodel consists of individual modules for the individual aspects of networked ontologies. The main modules are: (1) a metamodel for the OWL ontology language, (2) a rule metamodel, (3) a metamodel for ontology mappings, and (4) a metamodel for modular ontologies. The metamodel is grounded by translations to specific logical formalisms that define the semantics of the networked ontology model.

Yet, for the change representation we initially focus on the first part, i.e. the representation of changes to OWL ontologies (more specifically, OWL 1.1 ontologies). We will not provide a full specification of the metamodel in this document, instead we restrict to a presentation of the main concepts needed to understand the change representation.

Ontologies and Axioms An OWL ontology is defined by a set of axioms, of which OWL 1.1 provides six different types. Additionally, an ontology has an ontology URI which defines it uniquely, a (possibly empty)

set of imported ontologies, and a set of annotations. Additionally, also the ontology's elements, the axioms, can be annotated.

Figure 3.1 shows the metamodel presentation for ontologies, its axioms and annotations as metaclasses. The association *ontologyAxiom* connects the ontology to its axioms, whereas the associations *ontologyAnnotation* and *axiomAnnotation* connect ontologies respectively axioms to their annotations. The class *Ontology* has an attribute to identify the ontology, *URI*, whereas the attribute *URI* of the class *Annotation* specifies the type of annotation.

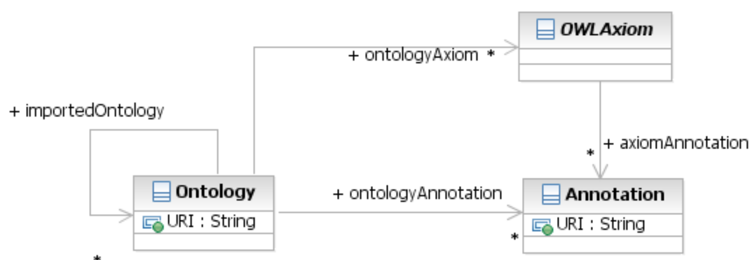

Figure 3.1: OWL metamodel: ontologies

Entities Entities are the fundamental building blocks of OWL 1.1 ontologies. OWL 1.1 has five entity types: data types, OWL classes[1], individuals, object properties and data properties. A datatype is the simplest type of data range. The second entity, a class, is a simple axiomatic class description classifying a set of instances. These class instances are called individuals and are also classified as OWL entities. At last, an object property connects an individual (belonging to a class) to another individual, whereas a data property connects an individual to a data value (belonging to a data range).

The OWL specifications highlight entities as the main building blocks of an OWL ontology and its axioms. Hence, the metamodel defines them as first-class objects in the form of metaclasses. Figure 3.2 presents an abstract metaclass *OWLEntity* which is defined as supertype of all types of entities. The five specific types of entities are specified as subtypes of *OWLEntity*: *Datatype*, *OWLClass*, *ObjectProperty*, *DataProperty* and *Individual*. An attribute *URI* of the abstract metaclass *OWLEntity* is inherited by all subclasses to identify the entity.

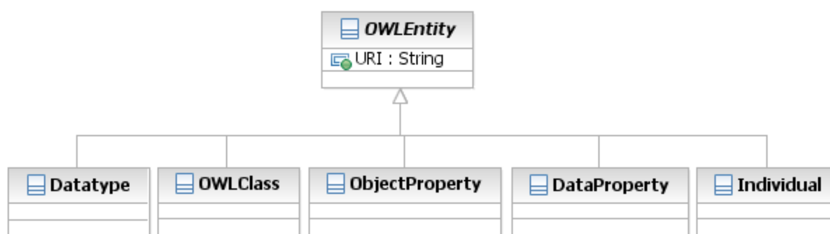

Figure 3.2: OWL metamodel: entities

Just like ontologies and axioms, also entities can be annotated. OWL categorizes such entity annotations as axioms. Hence an entity can be involved in two types of annotation: an annotation of the entity itself, or an annotation of such entity annotation as an axiom.

[1] OWL provides two classes with predefined URI and semantics: owl:Thing defines the set of all objects (top concept), whereas owl: Nothing defines the empty set of objects (bottom concept).

2006–2008 © Copyright lies with the respective authors and their institutions.

3.1.2 Change Ontology

A main requirement to support the editorial workflow (and in general the evolution of ontologies) is the formal and explicit representation of ontology changes in a suitable format. As we introduced in section 1.2.2, most of the existing approaches classify ontology changes in two levels: Elementary (atomic) and Composite although in [Sto04] the author introduces an additional level (complex).

According to the state of the art, an elementary (atomic) change is an operation that cannot be subdivided into smaller operations and that modifies only one entity of the ontology model. A composite change represent a group of elementary changes applied together that constitute a logical entity. Additionally, as we anticipated in section 1.3, existing approaches are dependent on the underlying ontology model (i.e. the are based on proprietary models (e.g. KAON) or for specific languages (e.g. OWL)) and consequently they have different set of elementary (atomic) changes.

Based on the previous notions, we propose a generic ontology for the representation of ontology changes (i.e. independent of the underlying ontology model) that models generic operations that are expected to be supported by any ontology language. For instance the operation "Add Class" should be supported by any ontology language although depending on the knowledge representation paradigm a class might be referred as a concept (operations at higher levels of abstraction (e.g. "Move Subtree") might even be referred the same independent of the knowledge representation paradigm). Additionally, our generic ontology defines some of its properties with an unconstrained range to avoid dependencies with any ontology language. The main idea behind our approach is that our generic ontology can be specialized for specific ontology languages while providing a common, independent model for the representation of ontology changes.

In the rest of this section, first we describe our generic ontology and then we will focus on our specialization for the OWL ontology language 1.1.

Compared to existing approaches, our proposed generic ontology propose a more fine-grained classification of ontology changes (i.e. atomic, entity and composite). We argue that even though the proposed elementary (atomic) changes are introduced as operations that cannot be subdivided into smaller operations, they are in all cases, considering the change at the entity level (i.e. concepts, properties, individuals), and in some cases they are not even minimal (e.g. include modify operations). Hence, we provide an additional lower level for the type of ontology change (i.e. atomic change) that represents the actual "atomic" operation that can be performed in a specific ontology model. The atomic change in our generic change ontology includes a property (with an unconstrained range) to associate it to the specific atomic elements. A specialization of our generic change ontology can then constraint the range of that property to the specific ontology language atomic elements. For instance, in OWL ontologies that lower level should be associated to the actual set of possible OWL axioms while in RDFS ontologies that lower level should be associated to the actual RDF triples. Having this lower level in the classification will provide a direct mapping between the user action and the ontology operations.

The next level in our classification is the entity level which allows us to associate the ontology changes to the ontology elements. As described in section 2, a key requirement to support the editorial workflow is to keep the state information about the ontology elements, which can be modified by the ontology editors (see section 3.3). Hence, knowing the state of the entity change, we will know the state of the associated ontology element. Similar to the atomic change, the entity change in our generic change ontology includes a property (with an unconstrained range) to associate it to the specific ontology elements. A specialization of our generic change ontology can then constraint the range of that property to the specific ontology language elements. To illustrate consider an ontology editor that inserts a new class (e.g. classX) to an OWL ontology. To represent this change, we will have to instantiate an entity change (e.g. Add class) and associate it with the actual class (e.g. classX). When assigning a state to the entity change, we will also know the state of the related ontology element (see section 3.3). As we can see from the previous discussion, our entity change corresponds to the elementary (atomic) change in the literature and therefore we can reuse and adapt existing proposals. Note that our generic ontology only provides a list of entity changes expected to be supported by every ontology language. However, as entity changes can be expressed by one or many

atomic changes, the *exhaustive* list of possible entity changes depends on the underlying ontology model and therefore it might only be represented in specializations of our change ontology.

Finally, similar to previous approaches, our composite change represent a group of changes applied together that constitute a logical entity (e.g. move a tree of classes, merge a set of siblings). It is evident, as it has been also noted in the state of the art, that is not possible to have an exhaustive list of composite changes (i.e. one can combine entity changes and composite changes in many different ways). Therefore in our ontology we provide only some of the most common operations.

Besides the taxonomy of ontology changes, our proposed ontology also models when was the change made, by whom and how it was performed (i.e. the exact sequence of changes). It is also important to note that our change ontology rely on many of the classes defined in the Ontology Metadata Vocabulary (OMV) which is a metadata schema that captures relevant information about ontologies (see [HP05]). Therefore it has been implemented as an OMV extension because it models specific ontology metadata information (i.e. ontology changes).

In the remainder of this section we will describe in detail the specialization of our generic change ontology for the underlying ontology model OWL DL 1.1[2]. Note that most of the description is also valid for the generic change ontology though, except for the OWL specific relations. The main classes and properties of the change ontology for OWL 1.1 are illustrated in Figure 3.3.

The class `Change` is the most important of our ontology and models the hierarchy of ontology changes reflecting the underlying ontology model by including all possible types of changes. Hence, it has three subclasses i.e. `AtomicChange`, `EntityChange` and `CompositeChange`. The atomic changes are further decomposed into additive changes (`Addition`) and removal changes (`Removal`). Following the approach proposed by [Kle04], we modelled each entity change operation as a class, and defined subsumption relations between these classes thus defining a hierarchy of entity operations. For example, all entity changes related to classes are grouped within the class `ClassChange` and in a similar way with the other types of ontology elements (e.g. properties, individuals). Note that the classes used to organize the entity operations are abstract classes that should not be instantiated. For the composite changes we provide only a number of classes that represent the most common composite operations. To capture the actual axiom applied to the ontology we associate the `AtomicChange` class with the `OWLAxiom` class from the OWL Object Definition Metamodel (OWL-ODM) defined in [HRW+06]. Similarly, the `EntityChange` class is associated to the `OWLEntity` class from OWL-ODM. Furthermore, to express that an entity change consists of one or more axiom changes, we associate both classes using the property `consistOfAtomicOperation` and to express that a composite change is a combination of other changes we define the property `consistOf`. Additionally to group all the changes made to a particular ontology version (see 3.2) we defined the class `changeSpecification` and relate it to the `Ontology` class from the OMV core to specify the previous and current version of the ontology before and after the changes. To specify who made a particular change, we relate the `Change` class with the `Person` class from the OMV core. Also, to keep the track of the actual sequence of changes (i.e. the order in which changes where performed) the property `hasPreviousChange` provide the required link between different changes. Finally, similar to [Sto04] we keep information supporting decision-making, such as cost, relevance and priority. The cost of a change determines the required effort to perform the change (e.g. number of derived operations necessary to complete the requested change). The relevance describes whether and how the change can fulfil the requirements. Consider for example a class deletion. The cost of that is estimated based on the ontology structure (e.g. the number of the subclasses and the total number of the individuals of these classes), so if the relevance is low and the cost is too high it would be better to avoid to perform that change unless it has a very high priority.

[2]The generic change ontology and the OWL 1.1 specialization are available at `http://omv.ontoware.org/2007/07/OWLChanges` and `http://omv.ontoware.org/2007/07/OWLChanges`, respectively

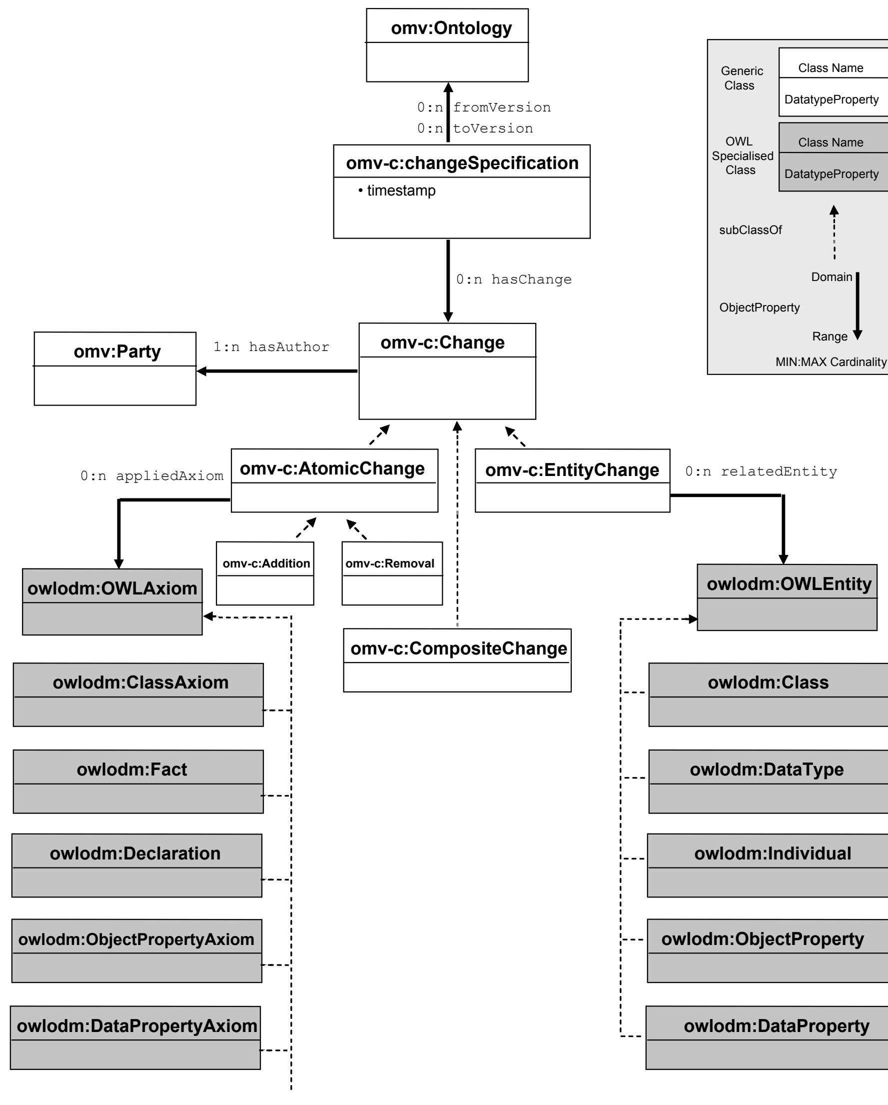

Figure 3.3: Change ontology for OWL 1.1

3.1.3 Ontology Change Capturing

Changes in ontologies need to be captured and stored in a certain format. After the definition of change ontology, we now have the standard to store changes about a certain ontology. The problem here is how to capture the changes in the certain ontology.

In the workflow for ontology editing, the editing activities are performed directly in the ontology editor interface. Thus, we need to track the user actions on the interface as ontology changing events. Then, we can store the ontology changing events and format them as structured data for future processing and propagation. This procedure is depicted in Figure 3.4.

As ontology editor interface has been implemented, we need to implement a monitor to track the user activities while they are using the interface to capture the changes. It is also important to realize the go-back functionality in the future. As the changes are applied to ontology while the ontology is saved, the ontology that describes the changes is also stored.

After creating the change ontology, there are three follow-up directions: (1) we can store and retrieve locally

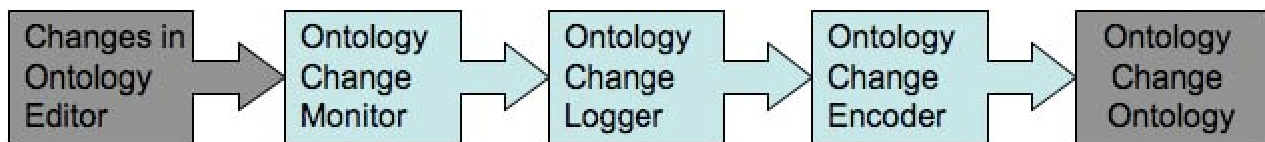

Figure 3.4: The procedure of change capturing

to see the previous changes; (2) we can propagate the change ontology via web service (3) it can be used for ontology versioning and conflict resolution components in the future prototypes.

3.2 Ontology Versioning

An important aspect that has to be taken into consideration when dealing with ontology changes is the versioning support. As we can see from section 1.2.1 there are different definitions and understandings of what is ontology versioning. So, instead of giving yet another definition, we provide here what we understand by ontology versioning: It is the mechanism in charge of the ontology change management which includes keeping track of the ontology changes and the ability to identify and maintain different variants of one ontology and their dependencies with support to undo/redo operations (e.g. rollback to a previous variant).

To keep the track of the ontology changes we generate a log that maintains the history (and order) of applied changes as a sequence of individuals of our proposed change ontology (see previous section). According to the classification of versioning given in [LM07], our approach is known as change(operation)-based versioning because we treat changes as first-class entities. The management of that ontology meta-information (i.e. applied changes) is responsibility of the ontology registry which is in charge of the administration of all the ontology related metadata (see below in this section). Furthermore, the information about the precise changes performed can be used to easily compute the difference between variants, or to implement a multiple undo/redo mechanism.

As we can see from the previous discussion, the issue of identifying the different variants (versions) of an ontology is not a trivial one. For example not even the OWL ontology language distinguishes between the notion of an ontology and a version of an ontology at all. So, even though ontologies are supposed to be identified by an URI, it may be that different versions of an ontology carry the same logical URI.

In general, a version is a variant of an ontology that is usually created after applying changes to an existing variant. Therefore we need a way to unambiguously identify the different versions as well as to keep track of the relationships between them. Based on [KF01], we consider that changes in ontologies are caused by: (i) changes in the domain; (ii) changes in the shared conceptualization; (iii) changes in the specification. Taking the definition of an ontology as a specification of a conceptualization, (i) and (ii) are semantic changes that lead to the creation of a new conceptualization, while (iii) is just a change in the representation of the same conceptualization (also known as a new revision) (e.g. updates of natural language descriptions of ontology elements). In any case, the change(s) result in a different physical representation of the ontology (i.e. different version). Consequently, it should be possible to identify each of those versions. Normally an ontology is identified by an URI, which according to [BLFM98] is a compact string of characters for identifying an abstract or physical resource. In [KF01] the authors propose that any version that constitutes a new conceptualization (i.e. changes of type (i) and (ii)) should have a unique URI, however in practice different versions of same ontology might share the same URI. Furthermore, even if a revision constitutes the same conceptualization of an ontology it is physically represented in a different file which might have additional metadata (e.g. updated ontology description, descriptions in different natural languages, different file location, etc.).

Currently however, many ontologies either do not provide any version information at all or the ontology editors explicitly do not want to change the version of the ontology after making some changes.

According to the editorial workflow requirements, only the validator should be able to change the version

2006–2008 © Copyright lies with the respective authors and their institutions.

information of an ontology when publishing it to the production environment. However, we should be able to identify the different versions of an ontology even if they are not published to keep the track of what version an ontology editor is working on i.e. consider for example the case when an ontology editor submit changes to an ontology version that has been approved but not published yet.

Therefore, based on the previous discussion we propose to identify a particular ontology using a composite identifier consisting of the logical identifier (URI) plus the version identifier. Note that since we are in a controlled environment we can assume that we will have always the version information. Furthermore we propose the following approach for the management of ontology versions to support the editorial workflow:

- The first stable version of an ontology (the populated ontology that satisfies all requirements in the step three of the ontology fishery lifecycle) becomes version 1. The version 1 of the ontology is considered an approved ontology (or published if a validator publishes it).
- From an approved or published ontology the first change to that version will automatically become a new version (i.e. with a different version information) (N+1), unless the editor explicitly specifies that the modified version will not become a new version (i.e. it will keep the same version information) (N). This version (the new one N+1 or the current modified one N) can receive many changes (by many ontology editors) without changing the version information until it becomes approved/published.
- When a validator wants to publish an ontology version, he can decide to keep the version information (i.e. do not change it) or specify another version information. In the former case, the version information will be the one chosen when the ontology was first modified (see above). In the latter case, the version information can be anything the user specifies (e.g. using a company versioning schema).

As we already mentioned before, the management of the change information is responsibility of the ontology registry. In our approach we rely on Oyster, the open source distributed ontology registry in NeOn. Oyster stores and manage OMV information which includes information about the change ontology introduced in the previous section. Oyster is designed to support different versions of ontologies (and at different locations), therefore ontologies are identified using a composite identifier (similar to what we explained above) that consists of the ontology URI plus the version information plus the resource location. The registry is crucial component in the infrastructure to support the editorial workflow: first the change capturing component monitors and captures changes from the ontology editors (see section 3.1.3). Those changes are formally represented using the change ontology described in section 3.1 and stored in Oyster which is in charge of the management of the different versions of the ontology. Also, using the registry functionalities, those changes will be searched and retrieved by the visualization component to show the differences between versions of the ontology (see section 3.6) and also to provide different views to ontology editors (depending on their role) where they can see the state of those changes (see section 3.3). Finally, the registry will be in charge of the propagation of those changes to the ontology related entities (see section 3.4).

3.3 Workflow Model for Collaborative Ontology Editing

Based on the analysis we presented in section 2 we found that some of the possible actions and states in the editorial workflow apply at different levels of abstraction. Therefore we refined and specialized the editorial workflow at two different levels: element level and ontology level. In general, the ontology state is automatically assigned based on the state of its elements as we describe below.

The specialized editorial workflow at the element level and ontology level are illustrated in Figure 3.5 and Figure 3.6 respectively. States are denoted by rectangles and actions by arrows.

The possible states that can be assigned to ontology elements are:

- *Draft*: This is the status assigned to any element when it pass first into the editorial workflow, or it is assigned to a single element when it was approved and then updated by a subject expert.

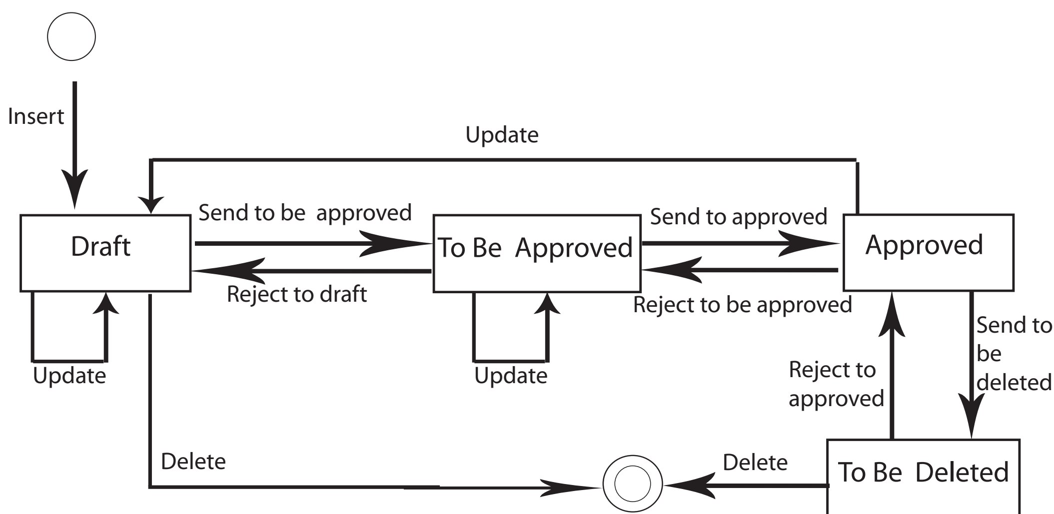

Figure 3.5: Editorial workflow at the element level

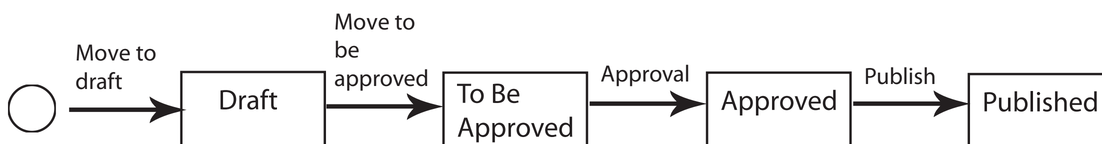

Figure 3.6: Editorial workflow at the ontology level

- *To be approved*: Once a subject expert is confident with a change in draft status and wants it to be validated, the element is passed to the "To Be Approved" status, and remains there until a validator accepts it or reject it.
- *Approved*: If a validator accepts a change in an element in the "To Be Approved" status, this element passes to the "Approved" status. Additionally, this is the default state for every element of the initial version of a stable ontology.
- *To be deleted*: If a subject expert considers that an element needs to be deleted, the item will be flagged with the "To Be Deleted" status and removed from the ontology, although only a validator would be able to definitively delete it.

The ontology has a state that is automatically assigned by the system (except from the "published" state) based on the state of its elements i.e. the state of an ontology is given by the "lower" (less stable) state of any of its elements. For instance it can happens that in the same ontology there are elements in "draft" state and elements in "to be approved" state, then the state of the ontology should be "draft". In the following we provide a detailed explanation:

- *Draft*: Any change send to an ontology in any state automatically send it into draft state.
- *To be approved*: When all changes to an ontology version are in "To Be Approved" state (or deleted) the ontology is automatically send to "To Be Approved" state.
- *Approved*: When all changes to an ontology version are in "Approved" state (or deleted) the ontology is automatically send to "Approved" state. Additionally, this is the default state of the initial version of a stable ontology.

2006–2008 © Copyright lies with the respective authors and their institutions.

- *Published*: Only when the ontology is in "Approved" state, it can be send by a validator to "Published" state.

As it was described in section 2, according to the lifecycle of the fisheries ontologies, the editorial workflow starts after getting a populated ontology that satisfies all requirements and that is considered stable. Hence, we assume that the initial state of this stable ontology (and all its elements) is "Accepted".
During the editorial workflow, actions are performed either:

- *Implicitly* e.g. when a user updates an element he does not explicitly performs an update action. In this case it has to be captured from the user interface. Action is recorded when the ontology is saved.
- *Explicitly* e.g. Validators explicitly approve or reject proposed changes. Action is recorded immediately when the user explicitly performs the action.

The actions that an ontology editor is allowed to perform depends on its role. Table 3.1 summarizes the actions that subject experts and validators respectively can perform at the element level, along with the state of the element before and after the execution of the action.

Table 3.1: Editorial workflow actions at the element level

Action	Previous State	Next State	Role	Remarks
Insert	—	Draft	SE	This action trigger the start of the workflow
Update	Draft	Draft	SE	—
Delete	Draft	—	SE	The item will be automatically deleted.
Send to be approved	Draft	To Be Approved	SE	This action moves the responsibility on the item from the subject expert to the validator. While an item is in the To be approved status the subject expert cannot modify it.
Reject to draft	To Be Approved	Draft	V	The validator reject the change
Update	To Be Approved	To Be Approved	V	An update done by a validator does not need to be double checked by other validators, so the element will remain in the same status as it was
Send to approved	To Be Approved	Approved	V	The validator accept it the change
Update	Approved	Draft	SE	This action trigger the start of the workflow
Send to be deleted	Approved	To Be Deleted	SE, V	—
Reject to be approved	Approved	To Be Approved	V	The validator wants the change to be reviewed again
Update	Approved	Approved	V	An update done by a validator does not need to be double checked by other validators, so the element will remain in the same status as it was
Reject to approved	To Be Deleted	Approved	V	If the validator does not agree with an element previously proposed To be deleted
Delete	To Be Deleted	—	V	The element is removed from the ontology

Table 3.2 describes the possible actions at the ontology level, along with the state of the ontology before and after the execution of the action. Note that, as we described above, most of these actions are automatically perform by the system (shown as "–" in the column "Role").

3.3.1 Workflow ontology

As we have shown in the previous discussion, the editorial workflow has a tight relationship with the operations performed to the ontology itself (i.e. the changes performed by ontology editors). Following the

Table 3.2: Editorial workflow actions at the ontology level

Action	Previous State	Next State	Role	Remarks
Move to draft	—	Draft	–	This action should be performed automatically by the system when the first change is submitted to an ontology version in "Approved" or "Published" state
Move to be approved	Draft	To Be Approved	–	This action should be performed automatically by the system when all changes to an ontology version are in "To Be Approved" state (or deleted)
Approval	To Be Approved	Approved	–	This action should be performed automatically by the system when all changes to an ontology version are in "Approved" state (or deleted)
Publish	Approved	Published	V	The validator can decide to publish an ontology only if it is in "Approved" state (i.e. copy the Approved ontology into the production environment and probably change version information)

approach in section 3.1 where we introduced our proposed change ontology, we developed a workflow ontology for the formal and explicit representation of the editorial workflow. Having both models (i.e. ontology changes and workflow process) formalized as ontologies will facilitate the representation of the tight relationship that exists between both of them.

The main classes and properties of the workflow ontology along with the relationships with the other ontologies in our approach (e.g. OMV and change ontology) are illustrated in Figure 3.7.

2006–2008 © Copyright lies with the respective authors and their institutions.

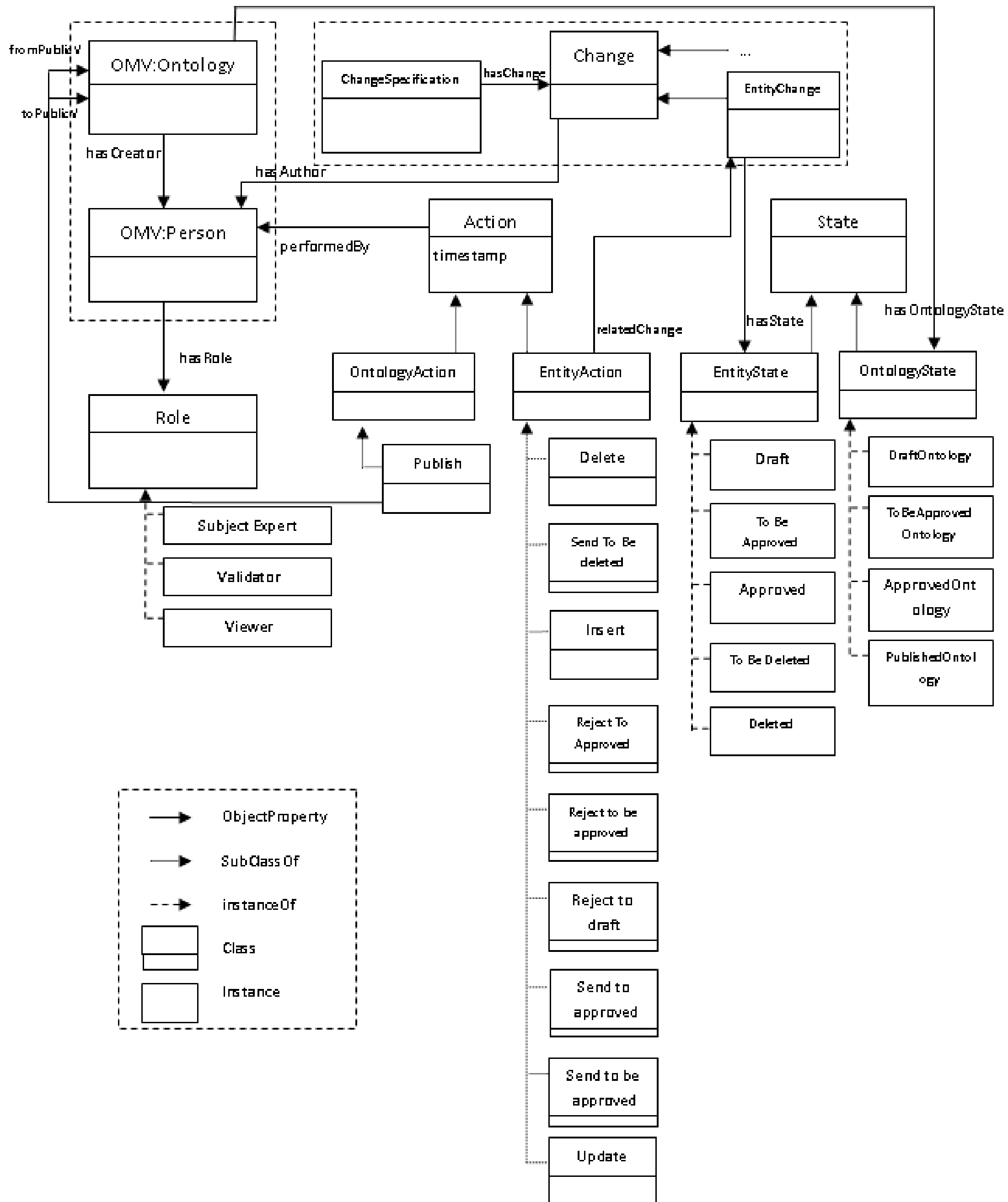

Figure 3.7: Workflow ontology

The different roles of the ontology editors are modeled as instances of the `Role` class that is related to the `Person` class of the OMV core ontology (i.e. a person has a role). To model explicitly the separation between the possible states of ontology elements (i.e. classes, properties and instances) and the possible states of the ontology itself, the `State` class is specialized in two subclasses (i.e. `EntityState` and `OntologyState`. Similarly to the roles, the possible values of the states are modeled as instances of their respective subclass. Furthermore, the two subclasses of `State` allow to represent the appropriate relationships at the element and ontology level: to specify that an ontology element has a particular state we rely on the class `EntityChange` from the change ontology that is associated to a particular ontology element (as described in section 3.1) and associate it with subclass `EntityState` and to specify that an ontology has a particular state we relay on the class `Ontology` from the OMV core and associate it with the subclass `OntologyState`. Note that the state is not assigned to the actual object (e.g. OWL ontology element or OWL ontology) but to the referring metadata entity (e.g. entity change or the ontology metadata) for two reasons: First, it is the actual referring entity the one that modifies the state of the object and second all this information is managed by the ontology registry that stores all related ontology metadata but not the ontologies themselves.

Finally, for the actions there is also a separation between the possible actions at the element level and actions at the ontology level. Hence, the `Action` class is specialized in two subclasses (i.e. `EntityAction` and `OntologyAction`. To track the whole process (and keep the history) of the workflow, the possible actions are modelled as subclasses of the appropriate `Action` subclass. Similar to the states, the two subclasses of `Action` also allow to represent the appropriate relationships at the element and ontology level: to specify that an action was performed over a particular ontology element, the subclass `EntityAction` is associated with class `Entitychange`. As we explained before, actions at the ontology level are performed automatically by the system except from publish which changes the public version of the ontology. Therefore, the only subclass of `OntologyAction` is `Publish` that is associated to the class `Ontology` to specify the previous and next public version of the ontology.

3.3.2 Visualization of Ontologies in the Editorial Workflow

According to the requirements, ontology editors (i.e. subject experts and validators) should be able to have different views of the ontologies in the editorial workflow depending on their role. We identify the following four views:

- *Draft view*: It shows the current approved ontology (i.e. all approved elements) plus all changes to that ontology version with difference between the two states clearly visible. The changes of the current editor are editable while changes from other editors are non editable.
- *Approved view*: It shows the current approved ontology (i.e. only approved information).
- *To Be Approved view*: It shows the current approved ontology (i.e. approved information) plus all changes (from all editors) pending to be approved with difference between the two states clearly visible.
- *To Be Deleted view*: It shows the current approved ontology (i.e. approved information) plus all suggested deletions (from all editors) with difference between the two states clearly visible.

Subject experts can visualize the first two views while validators can visualize the last three.

Illustrative example Lets consider the following simple scenario:

Subject expert expertFAO adds class1 to ontologyFAO_V1[3] and decides to create a new version of that ontology (i.e. V2) (which has automatically *Draft* state). As it is the first change to an ontology version, an instance of the class *ChangeSpecification* is created (e.g. changeSpecification1) to group all changes to that

[3] Assume the V1 is the first stable version of that ontology and hence has an *Approved* state

2006–2008 © Copyright lies with the respective authors and their institutions.

version. Additionally, to log the corresponding change, the particular instance of the *Change* class is created (e.g. addClass1) (with an associated *Draft* state) and associated to changeSpecification1 and to the actual class added (i.e. class1). Finally the corresponding action (e.g. insert1) is created and associated to the corresponding change.

```
<owlaxiomchange:ChangeSpecification rdf:ID="changeSpecification1">
 <owlaxiomchange:hasChange rdf:resource="#addClass1"/>
 <owlaxiomchange:fromVersion rdf:resource="#ontologyFAO_v1"/>
 <owlaxiomchange:toVersion>
  <omv:Ontology rdf:ID="ontologyFAO_v2">
   <omv:version rdf:datatype="http://www.w3.org/2001/XMLSchema#string">2.0</omv:version>
   <hasOntologyState> <OntologyState rdf:ID="DraftOntology"/> </hasOntologyState>
   ...
  </omv:Ontology>
 </owlaxiomchange:toVersion>
 ...
</owlaxiomchange:ChangeSpecification> <Insert rdf:ID="insert1">
 <relatedChange>
  <owlaxiomchange:EntityChange rdf:ID="addClass1">
    <owlaxiomchange:relatedEntity rdf:resource="#class1"/>
    <hasState> <EntityState rdf:ID="Draft"/> </hasState>
    <owlaxiomchange:hasAuthor>
     <omv:Person rdf:ID="expertFAO">
      <hasRole rdf:resource="#SubjectExpert"/>
     </omv:Person>
    </owlaxiomchange:hasAuthor>
    ...
  </owlaxiomchange:EntityChange>
 </relatedChange>
 <performedBy rdf:resource="#expertFAO"/>
 ...
</Insert>
```

Once expertFAO is sure of his change, he sends it to be approved (creates a new instance of the action *send to be approved*). Eventually, the user validatorFAO will analyse the proposed change and he will approve it (creates instance of action *send to approved*) or reject it (create instance of action *reject to draft*). The following is an extract of the approved action [4]:

```
<SendToApproved rdf:ID="approve1">
 <relatedChange rdf:resource="#addClass1"/>
 <performedBy>
  <omv:Person rdf:ID="validatorFAO">
    <hasRole rdf:resource="#Validator"/>
  </omv:Person>
 </performedBy>
 ...
</SendToApproved>
```

3.4 Change Propagation

One of the prerequisites for the realization of the Semantic Web vision is efficient knowledge sharing and reuse. In this scenario, ontologies are often reused by other ontologies, applications are increasingly relying on ontologies, ontology mappings are being created to integrate heterogenous knowledge bases, ontology instances can easily become a large number, etc. Furthermore, to achieve this goal, ontologies should be represented, described, exchanged, shared and accessed based on open standards such as the W3C standardized web ontology language OWL. Consequently, ontology-specific metadata becomes crucial to provide a basis for an effective access and exchange of ontologies across the web.

Therefore we should not consider an ontology as an independent and isolated entity, but rather we should consider the network of complex relationships and dependencies that an ontology can have. So, in our context of ontology changes, a change in an ontology does not only affect the ontology itself, but also to all its related artifacts (e.g. ontology instances, dependent ontologies, applications, mappings, metadata, etc.).

The issue of ontology change propagation has not been fully investigated and the relevant works in this area only address it partially 1.2.4. In [CAM+07], the author considers only the propagation of ontology changes

[4] Note that the corresponding change will have now *approved* state and since there are no other changes in this scenario to ontologyFAO_V2, its state will also change to *approved*.

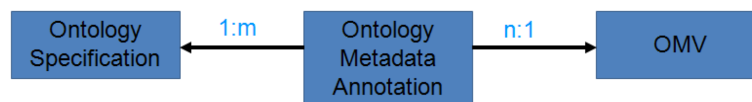

Figure 3.8: Conceptual model for the Ontology Metadata Annotation

to dependent ontologies, applications and instances. In fact, the author gives some general guidelines of how to treat the issue and only the propagation to dependent ontologies is further elaborated.

In the remainder of this section we will focus on one of the aspects that has not even yet considered: the propagation of ontology changes to its related metadata. Moreover we are considering a distributed scenario where the metadata is not centralized in one registry but distributed across many nodes. As we mentioned above, in our approach we are relying in the distributed ontology registry, Oyster.

3.4.1 Ontology Metadata

As we mentioned above, ontology-specific metadata becomes crucial to provide a basis for an effective access and exchange of ontologies across the web. In NeOn, we are using the standard proposal OMV[HP05] as the schema to model the metadata for networked ontologies (i.e. annotate ontologies (and related artifacts)).

Ontology metadata annotations are entities associated with a particular representation of an ontology, and that comply with a shared interpretation of the metadata (also known as knowledge entity) (e.g., ontologies, rule sets, controlled lists of terms). For instance, OMV defines the semantic interpretation for the ontology metadata and is represented as an ontology itself. In this case, a metadata annotation is an OMV instance that is associated to the corresponding ontology (identified by an URI (plus the version and location) (see [HP05])) and that obviously comply with the OMV schema (see Figure 3.8). Following the approach in [CAM+07] we propose to treat such metadata annotations as resources themselves (i.e. first class citizens) with their own lifecycle.

The metadata about ontologies is a very dynamic entity that can change throughout the time: First, ontologies themselves are continuously evolving. Consider the ontology as the formal, explicit specification of a shared conceptualisation [SBF98], it is clear that ontologies might change whenever one of the elements of the definition change (as we already introduced in section 3.2. For instance, domains are neither static nor fixed, they might evolve e.g. because a non existing element becomes part of the domain or because some elements become obsolete. A similar situation occurs with the shared conceptualization that might change during the time e.g. because the domain experts involved in the modeling of the ontology acquire additional knowledge about the domain. Finally, the specification might change e.g. because new ontology languages or new versions of the existing ones become available. Second, the semantic interpretation for the metadata (e.g. the OMV) might also change. OMV, as we described above is an ontology itself whose domain are the ontologies (i.e. it is meta-ontology), and therefore it might change for the same reasons explained before. Finally, the metadata itself can change i.e. it might need to be corrected or complemented with additional information. Consider, for example, an automatic tool that extracts and creates metadata which gets additional information or that a human expert adds additional information.

In any case, the aforementioned situations can lead to an invalid metadata. In the first case when the annotated ontology changes, it will either become a new version (i.e. with a different version information) or if specified by the editor remain the same version (i.e. with the same version information). In the former case, a new metadata annotation should be created, probably using information from the metadata annotation from the previous ontology version (i.e. note that in this case the metadata annotation associated to the previous ontology version does not become invalid). In the latter case, the new version should be properly re-annotated with updated metadata. Similarly, if the semantic interpretation of the metadata changes every metadata annotation should be reevaluated to determine if they are still valid.

2006–2008 © Copyright lies with the respective authors and their institutions.

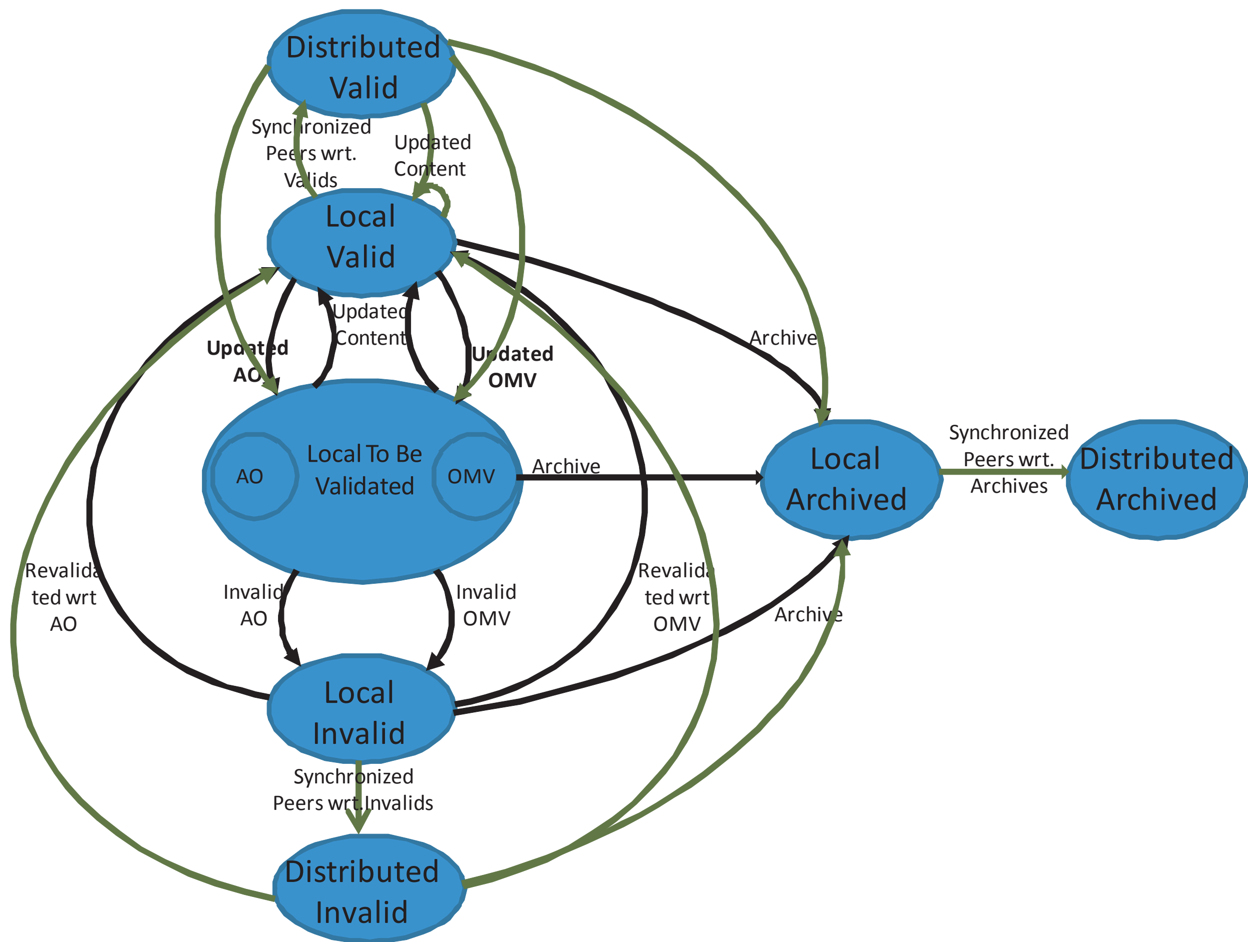

Figure 3.9: Ontology Metadata Annotation Lifecycle

3.4.2 Ontology Metadata Lifecycle

The ultimate goal of propagating ontology changes to metadata is to keep the metadata annotation up-to-date. Since we are considering ontology metadata annotations as first class entities, this task includes the definition of an appropriate model for the management of the lifecycle of the ontology metadata annotations, managing the evolution and change of metadata and ontologies in distributed contexts, and synchronizing adequately the evolution of all these related entities by means of notification mechanisms

Based on [CAM+07] and [MAC+06] we define a state machine which defines a set of states and transitions associated to it, for the management of the metadata lifecycle. In particular we extended and customized the proposed model for two reasons: First, we are considering a specific kind of metadata (i.e. ontology metadata). As proposed by the authors, the basic model can be extended by introducing sub-states, resulting in finer-grain definition of the behavior of specific types of metadata. Second, the proposed model is considering a centralized setting where all the metadata is stored in a central registry. In our scenario the ontology metadata will be distributed and managed by an appropriate distributed registry.

The proposed (modified) state machine diagram is presented in Figure 3.9.

In the following explanation, the ontology metadata annotation (i.e. the OMV instance) is denoted by OI. The ontology annotated by that instance is denoted by AO_{OI} and since the knowledge entity in our case is only OMV, we denoted it by OMV. Similar the content (payload) of the metadata annotation is denoted by $content_{OI}$.

Initially when the ontology metadata annotation is created (i.e. an OMV instance), it is in *Local Valid* state because it is originally created in one (any) local node. The possible state transitions events are the following:

- Changes in the described ontologies, i.e. by $AO_{OI} \rightarrow AO'_{OI}$
- Changes in the OMV, i.e. $OMV \rightarrow OMV'$
- Changes in the metadata annotation content, i.e. $content_{OI} \rightarrow content'_{OI}$

The first two previous events cause the transition of the ontology metadata annotation to either one of two possible *To Be Validated* state: *To Be Validated AO* or *To Be Validated OMV*. These are transitory state in which the ontology metadata annotation is awaiting to be validated. The validation process is usually an automated procedure that updates (if possible) the contents of the annotation and which results in a decision as to whether the updated ontology metadata annotation represents a new valid annotation i.e. describes correctly the current version of the annotated ontology and comply with the current OMV schema. Changing the metadata annotation content should not invalidate it. Therefore, this action does not change the state. In the following we explain in detail the two possible transitory states:

- For the *To Be Validated AO*, such procedure determines whether the existing annotation describes correctly the updated version of the annotated ontology or if the annotation content can be automatically updated accordingly or not. Consider for example the editorial workflow described in section 3.3, when an ontology version is approved (or published) leading to an update in the current version of the ontology, we have to verify if the associated metadata still describes correctly the ontology version or if it can be automatically updated to reflect the changes (e.g. Adding a new class to the ontology will lead to update (at least) the *numberOfClasses* property in the metadata annotation from N to N+1).
- For the *To Be Validated OMV*, the procedure should determine whether the new version of OMV can still be used to interpret the old metadata (i.e. it is still a valid OMV instance) or if the annotation content can be automatically updated accordingly or not.

The decision taken in any of the *Local To Be Validated* state can send the metadata annotation back to a *Local Valid* state (i.e. when it is still a valid annotation or the automatic process could update it accordingly), or to a *Local Invalid* state (i.e. when it is not a valid annotation and the automatic process could not update it accordingly).

When the metadata annotation is in *Local Invalid* state it could, however, go back to the *Local Valid* state after a manual re-validation process with respect to the annotated ontology or to the OMV schema (or both), depending of which one cause the metadata annotation to become invalid. In the case the annotated ontology changed and the metadata could not be updated accordingly, an ontology editor can manually update the content of the metadata annotation accordingly. Similarly, if the OMV changed and the automated process failed to update the metadata annotation accordingly, an ontology editor can manually re-validate it.

The *Local Archived* state indicates that a metadata annotation is still available for inspection, but it is not active (i.e. is not part of the metadata annotations used by the registry for searching). Usually this state allows to obtain "snapshots" of the metadata annotations available in a certain time.

One rare possible situation is that the ontology becomes obsolete/invalid and consequently unavailable/destroyed i.e. $AO_{OI} \rightarrow \emptyset$. In this case, the validation procedure is always assumed to fail, leading to a invalid state (first *Local Invalid* and after synchronization *Distributed Invalid*). From there, the metadata annotation is usually removed from the active set of metadata annotations and sent to be archived (i.e. $Archive(OI)$) or destroyed (i.e. $OI \rightarrow \emptyset$) e.g. when we do not want to save a copy of that information anymore.

The remaining states of our proposed state machine deal with the distributed environment we are addressing. The *Distributed Valid* state indicates that the metadata annotation is valid in every node of the distributed environment that stores that annotation (i.e. all nodes are synchronized). For example when a metadata annotation is created, it is created in one (any) node and becomes *Local Valid*. After nodes are synchronized and

2006–2008 © Copyright lies with the respective authors and their institutions.

the metadata annotation becomes available (and valid) in all the appropriate nodes, the metadata annotation becomes *Distributed Valid*. From the *Distributed Valid* state, we have the same possible state transitions events as in the *Local Valid* state i.e. $AO_{OI} \rightarrow AO'_{OI}$, $OMV \rightarrow OMV'$ and $content_{OI} \rightarrow content'_{OI}$. The first two events cause the same transition of the metadata annotation as described above. The only difference is when changing the metadata annotation content which in this case cause the transition of the metadata annotation to *Local Valid* state. The reason is that even if the content update does not invalidate the metadata annotation, it is performed in one (any) node and this change has to be again synchronized with all corresponding nodes.

The *Distributed Invalid* state indicates that the metadata annotation is invalid in every node of the distributed environment that stores that annotation (i.e. all nodes are synchronized). Similar to the previous situation, initially a metadata annotation becomes *Local Invalid* in one (any) node, and only after nodes are synchronized, it will become *Distributed Invalid*. From this state, the metadata annotation could go back to the *Local Valid* state after a manual re-validation process with respect to the annotated ontology or to the OMV schema (or both) as described above. Note that we are going back to the *Local Valid* and not to the *Distributed Valid* state due to the same reasons explained above.

Finally, in a similar way, the *Distributed Archived* state indicates that the metadata annotation is archived in every node of the distributed environment that stores that annotation (i.e. all nodes are synchronized). Again, initially a metadata annotation is *Local Archived* in one (any) node, and only after nodes are synchronized, it will become *Distributed Archived*.

Table 3.3 summarizes the events, state transitions and validation results (when applicable) of the state machine.

Table 3.3: Events, state transitions and validation actions

Event	State before event	State after event	State after validation
$AO_{OI} \rightarrow AO'_{OI}$	Local Valid	To Be Validated AO	Local Valid/Local Invalid
$AO_{OI} \rightarrow AO'_{OI}$	Distributed Valid	To Be Validated AO	Local Valid/Local Invalid
$OMV \rightarrow OMV'$	Local Valid	To Be Validated OMV	Local Valid/Local Invalid
$OMV \rightarrow OMV'$	Distributed Valid	To Be Validated OMV	Local Valid/Local Invalid
$content_{OI} \rightarrow content'_{OI}$	Local Valid	Local Valid	N/A
$content_{OI} \rightarrow content'_{OI}$	Distributed Valid	Local Valid	N/A
$AO_{OI} \rightarrow \emptyset$	N/A	To Be Validated AO	Local Invalid
$OI \rightarrow \emptyset$	N/A	N/A	N/A
$Archive(OI)$	N/A	Local Archived	N/A
$Revalidate(OI)$	Local Invalid	Local Valid	N/A
$Revalidate(OI)$	Distributed Invalid	Local Valid	N/A
$Synchronization$	Local Valid	Distributed Valid	N/A
$Synchronization$	Local Invalid	Distributed Invalid	N/A
$Synchronization$	Local Archived	Distributed Archived	N/A

3.4.3 Propagation of Ontology Changes to Ontology Metadata

Relationship between the editorial workflow and the ontology metadata lifecycle Once we have defined an appropriate model for the management of the lifecycle of the ontology metadata annotations, the next step to keep the annotations up-date is the management of the changes in ontologies and metadata in distributed contexts (i.e. how those changes are related). As we introduced in section 2, the central procedure for the ontology change management is the editorial workflow which allows to control how and when an ontology changes. Hence, our goal is to clarify how that workflow is related to the lifecycle model for metadata annotations.

Figure 3.10 illustrates the relationship that exists between the editorial workflow described in section 3.3 and the state machine described in the previous section (see Figure 3.9).

Whenever an ontology version is approved (or published) either a new version of the ontology is created (i.e. version information is changed from N to N+1) or the same ontology version is updated (i.e. the version information remains the same) (see section 3.2).

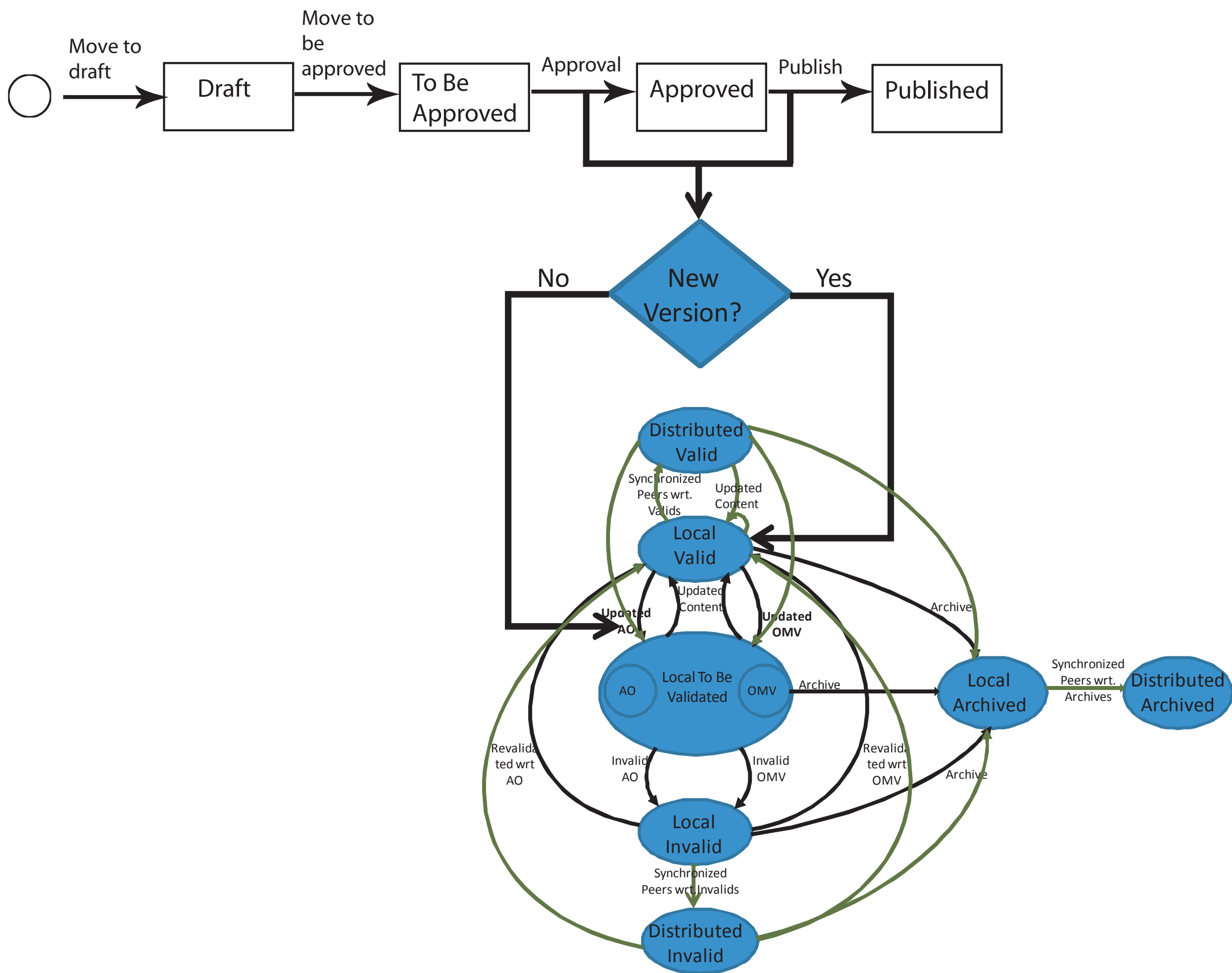

Figure 3.10: Editorial Workflow & Metadata Lifecycle

If a new version is created, then a new metadata annotation should be created (i.e. OI_{N+1}. The metadata annotation associated to the previous ontology version (i.e. OI_N remains the same because the corresponding ontology did not change. Note that the OI_{N+1} might be created reusing the information from OI_N. As we explained above, after creating OI_{N+1}, it will be in *Local Valid* state.

When the version information remains the same after some changes in the ontology (i.e. event $AO_N \rightarrow AO'_N$), the corresponding metadata annotation (i.e. OI_N) should be sent to the *Local To Be Validated* state as described above.

Notification & Synchronization As we introduced in the previous section, the changes in an annotated ontology that results in an updated content of the same ontology version (i.e. $AO_N \rightarrow AO'_N$ event) may affect its related metadata (i.e. OI_N). Consequently, those changes should be propagated to the corresponding metadata annotation.

Since we are considering a distributed environment, we propose to decompose the problem in two steps: First, Ontology metadata annotations should be notified whenever a change occurs in the corresponding ontology. The notification automatically sends the metadata annotation into the *To Be Validate AO* state in which it will be evaluated if it is still a valid annotation or if it is possible to automatically update it accordingly (see previous section). If is not possible to automatically update the annotation, it will become invalid even though

2006–2008 © Copyright lies with the respective authors and their institutions.

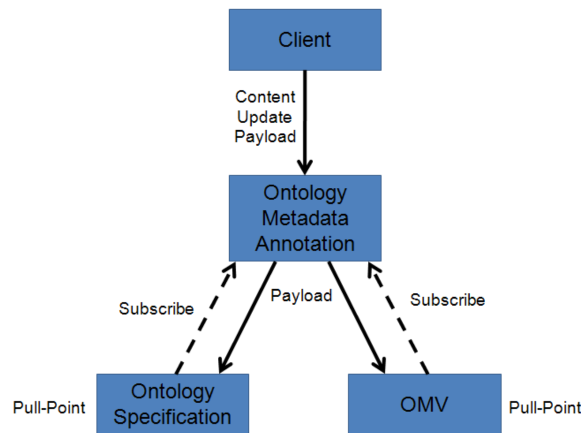

Figure 3.11: Notification of Ontology Metadata Annotations changes

after a manual intervention it might become valid again. Second, since we are considering a distributed environment, the annotations should be synchronized. Independently of the state of the annotation (i.e. if the annotation became valid after its content was updated (automatically or manually) or if the annotation became invalid), every node in the network aware of that annotation should have the same information.

For the first step, following the approach in [CAM$^+$07] we propose to use a set of notification mechanisms based on WS-Notification. Therefore, we propose a pre-defined topic associated to ontology changes. In Figure 3.11 we illustrate how the Ontology Metadata Annotation (OI) is subscribed to notifications of change in its corresponding ontology[5] (specified by their last modification time), which will probably make it change its state.

When the OI receives a notification, it will automatically change its state to *Local To Be Validated* and it will activate a re-validation procedure that will perform the following operations:

- Validate the content of the OI with respect to the changes in the corresponding annotated ontology.
- If validation determines that the OI is valid (i.e. the changes in the ontology did not affect it), then the OI is sent to the *Local Valid* state.
- If validation determines that the OI is invalid (i.e. the changes in the ontology did affect it), then an automatic process of re-annotation is triggered. After the process finishes, the OI is again validated.
  - If validation determines that the OI is valid (i.e. the OI could be updated successfully by the automatic process), then the OI is sent to the *Local Valid* state.
  - If validation determines that the OI is invalid (i.e. the OI could not be updated successfully), then the OI is sent to the *Local Invalid* state.

When an OI is in the *Local Invalid* state, a manual re-annotation can still be performed that will make it change its state to *Local Valid* as explained previously.

One additional consideration for the notification mechanism is whether to use a push or pull approach. The benefits and disadvantages of both approaches has been already analyzed in many places (e.g. [Sto04]). Since we are considering a distributed environment for the management of ontology metadata annotations, we propose to use the WS-Notification mechanism with a pull-style notification [OAS06].

For the synchronization of the nodes in the distributed environment, we propose that periodically nodes will contact other nodes in the network to exchange updated information. For this task, each metadata annotation in the registry should have the information about the last modification time of the associated ontology version. Hence, during the the synchronization the following process occurs when node A contacts node B:

[5]The Figure also shows how the ontology metadata annotation is subscribed to notifications of changes in the OMV schema

- For every OI that is maintained in both A and B. If the metadata annotation maintained in node A associated to ontology X (A_{OI_X})has a lower modification time than the corresponding in B (B_{OI_X}), then update the local annotation. Otherwise do not do anything.

Note that only local annotations can be updated. If the contacted node has an out-of-date information, then we do not do anything as we should not modify remote information. Eventually, the remote node will contact a node with updated information and modify its own information accordingly.

3.5 Conflict Resolution and Concurrent Ontology Editing

We are considering two types of conflicts in the collaborative workflow: (1) logical conflicts in the form of inconsistencies and (2) conflicts due to concurrent editing of an ontology.

In the context of the collaborative workflow, logical inconsistencies are possible to occur due to modeling errors, or mutually inconsistent views of the editors on the domain. There exist various approaches to resolve logical inconsistencies. In deliverable D1.2.1 [QHJ07] we have presented our approach to deal with inconsistencies in networked ontologies. We refer the reader to this deliverable for further details. It should be noted that logical inconsistencies can occur even in the case of a single user editing a single ontology.

The second type of conflict may occur when multiple users *concurrently* edit the same ontology, i.e. the ontology editors operate on the same life instance of the ontology (instead of local copies as assumed in the previous sections). For example, one editor may choose to update an ontology element that has just been deleted by another editor. A similar kind of problem is considered database research under concurrent updates to databases. While this field is well understood and transaction mechanisms are a commodity in databases, the approaches are not directly applicable to concurrent ontology modeling.

A simple means to support concurrent ontology editing is the provision of a central ontology store that can be accessed by multiple clients at the same time. A centralized architecture like this has a number of advantages:

- The synchronous update mechanisms allows ontology editors to always access, review, and modify the most recent (because sole) copy of an ontology.
- Chances of conflicts are greatly reduced due to the immediate distribution of changes to all clients.
- Users can be notified on a life basis of all changes that are made by other clients in a timely manner.
- By adding communication means like IRC (not implemented yet) to complement the formal collaboration, the collaborative ontology modeling experience can be greatly enhanced. Crucial changes can be discussed before they are actually committed.

The collaboration server approach also faces some challenges:

- The central ontology can only be modified while being on-line (i.e. the modifying client is connected with the collaboration server). Off-line modifications would detach the central ontology from any local copy.
- Even when real conflicts can be drastically reduced by life updates of all clients, it cannot be guaranteed that they do not occur, e.g. two different rename operations of the same class.
- Managing the collaboration of multiple ontology editors from a central server puts the workload of storing and distributing change to the central unit. This is only feasible for a limit number of clients and a decent amount of traffic.
- The network latency must be kept to a minimum in order to ensure a smooth modeling experience for ontology editors, which might not be feasible in a globally distributed network but is definitely feasible within the intranet or VPN of a company or other organization.

2006–2008 © Copyright lies with the respective authors and their institutions.

3.6 Ontology Comparison

Comparing ontologies is useful within an editing workflow to check if two versions of an ontology are the same, and in the case they are not, to highlight changes. This functionality can be seen as similar to the *diff* tool used in classical software versioning systems. However, while *diff* only compares software source code at a syntactic level, ontology comparison has to rely on the semantics underlying the ontology: it has to check whether two ontologies are semantically equivalent or not.

The detection of semantic equivalence between ontology leads to a number of theoretical and practical issues. First, a clear and practical formalization of the notion of semantic equivalence of ontologies has to be provided, together with a procedure implementing this definition. Second, dealing with the semantics of the ontology leads to the comparison of infinite structures, making the identification of a finite set of changes not practicable. Finally, this procedure for detecting semantic comparison is very complex and requires the use of a reasoner, while simple tests could lead to the conclusion that a semantic comparison is not required.

In this section, we describe our approach for semantic comparison that is based on several levels of comparison, including comparison of vocabularies, syntactic comparisons and semantic comparisons. Knowing the implications relating these different comparisons, it can then be detected at early stages, through simple tests, if a semantic comparison would be required, and simple changes could be captured in the change ontology (see Section 3.1).

3.6.1 Notations

This section introduces some notations and definitions used in the following.

Ontology: An ontology is a set of axioms, an axiom being a statement asserted on the considered entities (i.e. sub-class, equivalent, disjoint, instance-of, relations, etc.)

The set of axioms constituting an ontology describes a logical theory about objects of the domain and relations between these objects. These elements are represented by the entities of the ontology (classes, properties and individuals).

Vocabulary: The set of local names of entities used in the description of the axioms of an ontology is called its vocabulary (sometimes also called the signature of the ontology): it contains the building blocks of the expressions and axioms of the ontology. We note $V(O)$ the vocabulary of an ontology O. The vocabulary of an ontology can also be seen as its *namespace*: the set of names it defines.

There are basically two ways of expressing the semantics of an ontology (from a logical perspective): by considering the set of models of the ontology (in the sense of a model-theoretic semantics), or by considering its set of logical consequences. In most of the cases, these two ways would lead to the same results, so we have choosen to rely on the simplest one: logical consequences.

Semantics of an ontology: The set of logical consequences of an ontology O, noted $C(O)$, is the set of axioms that can be inferred (or entailed) from O. Basically, it is a set of axioms (including O) that can be proved to be true according to O and to the semantics of the employed representation language. It is worth to mention that, when using highly expressive languages like OWL, $C(O)$ is generally infinite.

3.6.2 Semantic Comparison

Definition and Procedure

On the basis of the previous definitions, the definition of semantic equivalence is quite straightforward.

Semantic comparison: Two ontologies are semantically equivalent if they express the same semantics, i.e. if they have the same set of logical consequences. Therefore, O_1 is said to be semantically equivalent to O_2 iff $C(O_1) \equiv C(O_2)$.
The procedure to detect equivalence relies on the notion of semantic inclusion, that is defined by $C(O_1) \subseteq C(O_2)$. It can be easily shown that, when using standard ontology representation languages, $C(O_1) \subseteq C(O_2)$ iff $O_1 \subseteq C(O_2)$, which can formulated as checking if O_1 is entailed by O_2. O_1 and O_2 are equivalent iff O_1 includes O_2 and O_2 includes O_1. Therefore, semantic equivalence can be detected by checking if the two considered ontologies entail each-other.

Practical Problems

The previous section provides a definition for semantic equivalence between ontologies and a procedure to detect this relation. However, in practice, this procedure raises a number of problems. First, we are not only interested in knowing whether or not an ontology is equivalent to another, but also in computing the difference between them, to be captured according to the change ontology. This difference is composed of two parts:

The axioms removed from O_1 defined by $C(O_1) \setminus C(O_2)$

The axioms added to O_2 defined by $C(O_2) \setminus C(O_1)$

The most obvious problem posed by this definition is that, as $C(O_1)$ and $C(O_2)$ are generally infinite, the two difference sets may be infinite as well, and therefore, can not be computed.
Another important problem concerns the complexity of the procedure to detect semantic equivalence. Testing if an ontology entails another one requires an large amount of inferences that are, at least for expressive languages like OWL, very complex and time consuming. However, simple tests can help us in avoiding most this complex procedures. Indeed, it does not seem to make much sense to check, for example, if a big ontology in the medical domain and a small one about digital cameras are semantically equivalent.
In the following, we describe different levels of comparison, that can be seen as either complete or correct approximations of the semantic comparison. By looking at the implications relating these levels, we provide an overall procedure where the need for a complex semantic comparison is detected at early stages and changes are highlighted at simpler levels.

3.6.3 Syntactic Comparison

The most obvious approximation of a semantic comparison as described previously is a syntactic comparison, where declared axioms of the ontology are compared instead of logical consequences.

Syntactic comparison: Two ontologies O_1 and O_2 are syntactically equivalent if they declare the same set of axioms, i.e. iff $O_1 \equiv O_2$. The syntactic difference between these two ontologies can then be computed through the two set differences, and the result of the comparison captured in the change ontology, either as added or removed axioms.
One difficulty concerning this comparison is that we want to abstract from the language used to encode the ontologies. The same axiom can be encoded differently in OWL and DAML+OIL for example. We solve this by introducing a set of manually defined “mappings” between axiom representations in different languages, and transforming the set of axioms of both ontologies into a common encoding, thanks to these mappings.

2006–2008 © Copyright lies with the respective authors and their institutions.

Relation with the semantic comparison: Two ontologies being syntactically the same are trivially semantically equivalent. Therefore, we can see the syntactic comparison as a correct approximation of the semantic comparison. This means that for ontologies to be syntactically equivalent implies to be also semantically equivalent. As the cost of the syntactic comparison is lower than the one of the semantic comparison, checking first at the syntactic level is useful to detect if a semantic comparison is needed. Also, in most of the cases, the sets of differences computed at the syntactic level are subsets of the sets of differences at the semantic level. There can however be exceptions to this, in particular when a removed axiom can still be inferred from the new version of the ontology.

3.6.4 Dealing with Anonymous Entities

One of the difficulties of manipulating ontologies concerns the use of complex expressions built using the language constructs. When visualizing an ontology for example, it is generally simpler to consider only the named entities, like the atomic concepts. In the same way, computing ontology relations by restricting the comparison to named entities is a natural way to simplify the procedure.

We define $N(O)$ as the set of axioms occurring between named entities in an ontology O. Obviously, $N(O) \subseteq O$. In the same way, $NC(O)$ refers to the set of logical consequences linking named entities only. Thus, $N(O) \subseteq NC(O) \subseteq C(O)$. It is worth mentioning that, while $C(O)$ is generally infinite, $NC(O)$ is generally finite.

Semantic comparison on named entities: Two ontologies O_1 and O_2 are said to be semantically equivalent with respect to named entities iff $NC(O_1) \equiv NC(O_2)$. In the same way, the differences between these two ontologies are calculated here in terms of the set differences between $NC(O_1)$ and $NC(O_2)$.

The advantage of using this comparison is that, since the logical consequences restricted to named entities is finite, the difference sets are also finite and can therefore be computed. However, the complexity of this comparison is still very high, in particular when employing a naive approach checking every possible axioms between named entities using a reasoner.

The differences detected using this comparison mechanism can be captured, in the same way as for the syntactic comparison, in terms of axiom level changes in the change ontology. However, the change ontology currently does not distinguish between changes appearing at the semantic level (the entailed axioms) and the ones appearing at the syntactic level only.

Relation with the semantic comparison: Since $NC(O) \subseteq C(O)$, the semantic comparison on named entities is trivially a complete approximation of the semantic comparison. Also, the difference sets in the case where we restrict to named entities are subsets of the difference sets without restriction.

Syntactic comparison on named entities: In a similar way, two ontologies O_1 and O_2 are said to be semantically equivalent with respect to named entities iff $N(O_1) \equiv N(O_2)$, and the differences are calculated here in terms of the set differences between $N(O_1)$ and $N(O_2)$.

Relation with the syntactic comparison: Like for the semantic comparison, the syntactic comparison on named entities is a complete approximation of the syntactic comparison.

3.6.5 Vocabulary Comparison

Finally, the simplest element that can be considered in an ontology is the vocabulary. Indeed, a necessary condition for two ontologies to be semantically equivalent is to share the same signature.

Vocabulary comparison: Two ontologies O_1 and O_2 are said to share the same vocabulary if $V(O_1) \equiv V(O_2)$. Therefore, the difference between these ontologies at the level of the vocabulary can be computed using the two set differences between $V(O_1)$ and $V(O_2)$. Depending on their type, the elements in these difference sets can be represented in the change ontology by changes of the form $removeClass(C)$ or $addProperty(p)$ for example.

Relation with other comparisons: Vocabulary comparison constitutes a complete approximation of all the other comparisons. This means that only ontologies that share the same vocabulary could be syntactically or semantically equivalent. It therefore constitutes a very simple test allowing to avoid the use of more complex comparisons in the many cases where ontologies are different at the level the vocabulary.

3.6.6 Overall Procedure

Looking at the complexity of the different comparisons described above, we can derive an overall procedure, computing these different levels of comparison in the most efficient order. Indeed, starting from the simplest, the vocabulary comparison, it is already possible to decide if the ontology would be semantically different, or if other comparisons are required to check if they are equivalent (because semantic equivalence implies vocabulary equivalence). If a vocabulary equivalence is detected, the procedure continues at the syntactic level and then, if required, at the semantic level, starting in both cases by the comparison on named entities only. Even if all the levels are computed, differences computed at each step constitute valuable information to capture the changes between the two ontologies.

2006–2008 © Copyright lies with the respective authors and their institutions.

Chapter 4

Implementation

The components introduced in the previous sections are planned to be implemented as proof-of-concept prototypes. Some of these components have been already implemented as plug-ins of NeOn Toolkit, and some others are planned in the next step of software development.

4.1 Overall Software Architecture

To implement the functions of workflow for ontology editing based on the requirements proposed in Chapter 2, we plan to develop and integrate the following plug-ins into the software prototype of workflow for ontology editing:

- OWL Ontology Editor is used to edit OWL ontologies within an intuitive GUI with support of visualizing status of ontology and elements of ontology in editing process;
- Ontology Versioning plug-in is designed to track different versions of ontologies based on ontology metadata about versions;
- Change Capturing plug-in aims to monitor the activities in OWL Ontology Editor and log the changes of ontologies in a structured change ontology;
- Conflict Resolution plug-in resolves the conflicting editorial activities from different users;
- Ontology Comparison plug-in finds out the differences between two ontologies if there is any;
- Change Propagation plug-in is responsible for distributing the up-to-date information about the ontology evolution cycle, including the versions, changes and status in the workflow;
- the Oyster aims to provide services for propagating information in distributed ontology registry as the role of ontology registry service; Oyster also supports representing the workflow model;
- KAON2 acts as an ontology reasoning engine for conflict resolution and OWL API provider;
- Datamodel plug-in offers infrastructure and APIs at background for handling networked ontologies.

By aligning to general NeOn architecture introduced in WP6 [GAGW06], we depict the software architecture of workflow for ontology editing related components.

Figure 4.1 represents the overall architecture of workflow for ontology editing related components, which are going to be implemented in first prototype and future prototypes step by step. We call the prototype introduced in this deliverable as the first prototype for ontology evolution and propagation, while it is also a part of the first prototype of fishery ontology life-cycle management system, which will be introduced in Task 7.4. We can see there are three layers in this architecture:

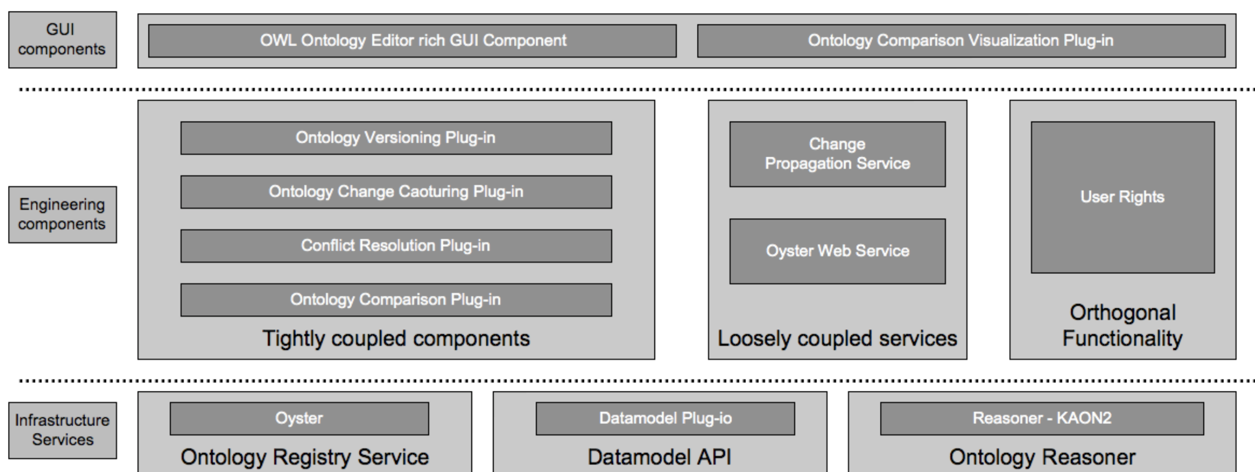

Figure 4.1: The overall architecture of workflow for ontology editing related components

- Schema Modeling in OWL Ontology Editor as central part in realizing user interface in the software, while we both provide rich and thin GUI components based on the requirement of software component itself.
- In NeOn architecture, we have identified the difference between loosely and tightly coupled engineering components. Apparently, Oyster and Change Propagation web services are classified as loosely coupled engineering components, and As we consider our components to be distributed over network, we implemented a distributed software component management infrastructure to coordinate all distributed software components.
- The updated Oyster ontology registry services supports the workflow related tasks with underlying ontology reasoner – KAON2.

However, we are not going to implement all functions at once. Basic components, such as Change Capturing plug-in, are fundamental requirements to implement other functions , such as ontology versioning. Obviously, Ontology Versioning requires capturing the ontology changes at first. Therefore, following components are included in the first prototype:

- OWL Ontology Editor without user rights control
- Change Capturing plug-in
- Change Propagation plug-in
- the Oyster aims to provide web services for propagating information in distributed ontology registry as the role of ontology registry service;
- KAON2 acts as an ontology reasoning engine and OWL API provider;
- NeOn distributed software component management coordinates these components in the distributed networking environment.

After presenting the architecture of prototype, we will still introduce all the individual plug-ins that are planned to be implemented in detail. The following table depicts the mappings among the requirements described in Chapter 2, the methods introduced in Chapter 3 and the actual prototypes addressed here.

2006–2008 © Copyright lies with the respective authors and their institutions.

Functional Requirements	Detailed Requirements	Methods	Prototypes
UC-9.1	Capture Changes Representing Changes	Change Ontology Change Ontology	Change Capturing Oyster
UC-9.2	Store Change History View Ontology Versions Compare Ontology Versions Propagate Changes	Registry API Ontology Versioning Ontology Comparison WS-Notification	Oyster Ontology Versioning Ontology Comparison Propagation Service
UC-9.3	View Use Statistics	Change Ontology	Oyster
UC-9.4	View Ontology Statistics Versioning	Change Ontology Ontology Statistics Ontology Versioning	Oyster Oyster Oyster
UC-15.1	Insert Ontology Element Update Ontology Element	Workflow Model	Schema Modeling
UC-15.2	Delete Ontology Element	Workflow Model	Schema Modeling
UC-15.3	Store Status Change Get Status Change Conflict Resolution	Change Ontology Workflow Model Conflict Resolution	Oyster Status Management Conflict Resolution
UC-15.4	Publish Ontology	Change Propagation	Change Capturing Propagation Service

Table 4.1: Mappings among requirements, methods and prototypes

4.1.1 OWL Ontology Editor

There are different components included in the OWL Ontology Editor. There are some components implemented, under development or planned for future realization. Table 4.1 tells us how the OWL Ontology Editor covers the UC-15.1: Edit ontology element, UC-15.2: Delete ontology element and UC-15.3: Change status of element, so we discuss how the OWL Ontology Editor implements these use cases.

OWL Ontology Editor has been implemented to support ontology modeling and maintenance in UC-15.1 and UC-15.2. Schema Modeling plug-in in OWL Ontology Editor contains the core UI components and connects them with the underlying datamodel via the com.ontoprise.ontostudio.datamodel plug-in. The main UI elements include the ontology navigator and the property editors for the different ontology entities. We also have the basic Eclipse SWT (Standard Widget Toolkit) classes are extended and customized to support the GUI plug-in.

To further extend the functionality of OWL Ontology Editor, we have implemented the Change Capturing plug-in and integrated it with OWL Ontology Editor. Currently, Oyster supports workflow model by processing metadata that records the activities in editing ontologies, by which Oyster also provides basis of Ontology Versioning functions. Moreover, following plug-ins are being developed to complete the workflow of ontology editing process. However, they are not part of the OWL Ontology Editor, they are independent plug-ins and interact with OWL Ontology Editor by providing services or APIs.

- Change Propagation plug-in propagates the changes that have been made on ontologies than are completed in editing procedure via web services. Users are able to track the changes of ontologies by accessing remotely.
- Ontology Comparison plug-in provides differences checking for ontologies by comparing the ontologies. This is also required for the Ontology Versioning plug-in to identify different versions.
- Conflict Resolution plug-in enables conflict-checking of ontologies in the editing procedure and resolve the potential conflicts semi-automatically or manually.

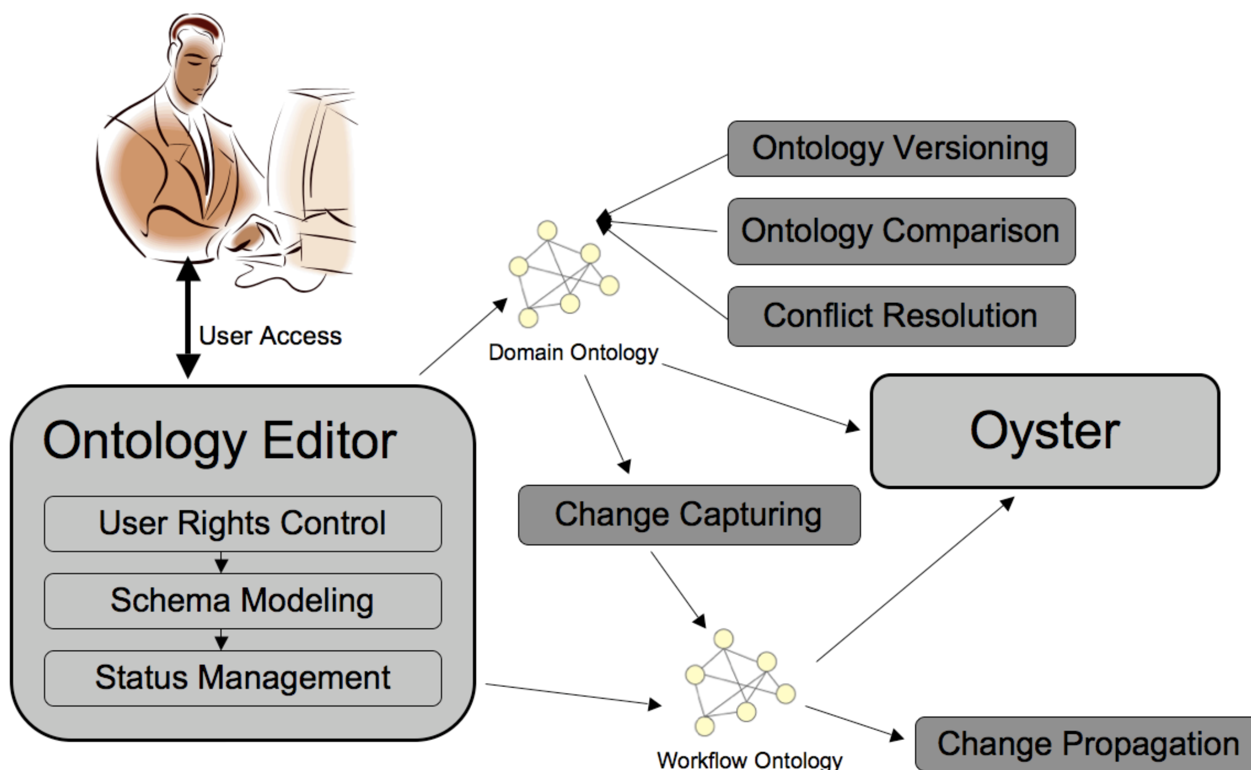

Figure 4.2: The OWL Ontology Editor and its related components

Extensions to OWL Ontology Editor to support editorial workflow To support UC-15.3, we plan to develop two plug-in as extensions to OWL Ontology Editor: One is Status Management plug-in, and the other is User Rights Control plug-in. They extends the OWL Ontology Editor plug-in. By recalling the Section 2.1 and 3.3, we can see the workflow consists of different status in ontology editing.

The representation of workflow provides different user groups the overall view on the status in editing ontology. The User Rights Control plug-in makes sure appropriate accessibility from different user groups. For example, the ontology editor does not have rights to approve and validate an ontology. Different user groups apparently have their own rights in accessing different status when editing ontologies. For example, ontology validator could approve whether an ontology is validated or need to be revised by ontology editor. Therefore, we need a user rights control component to monitor the accessibility of different user groups.

The structure of extensions with user rights controlling functionality are also depicted in Figure 4.2. In this figure, we can see while a user want to access the OWL Ontology Editor, the user rights control component checks the identity of this particular user, and workflow representation components authorize the appropriate functions for this user in the whole workflow of ontology editing.

Next, we introduce the plug-ins related to direct supporting propagation and change management as our focus in detail.

4.1.2 Change Capturing Component

In the previous chapter, we introduced the procedure for ontology Change Capturing. Here, we will discuss how this procedure incorporates to a real life application to support UC-9.1, UC-9.2 and UC-15.4.

With input of OWL Ontology Editor events, the system starts with a change capturer to track the changes in OWL Ontology Editor, and transfers these events into ontology changing logs via a logger component. Afterwards, the Oyster API encodes these changes into change ontology. As we proposed in previous chapter, local and remote access can be applied to the new created change ontology. The Figure 4.3 depicts

2006–2008 © Copyright lies with the respective authors and their institutions.

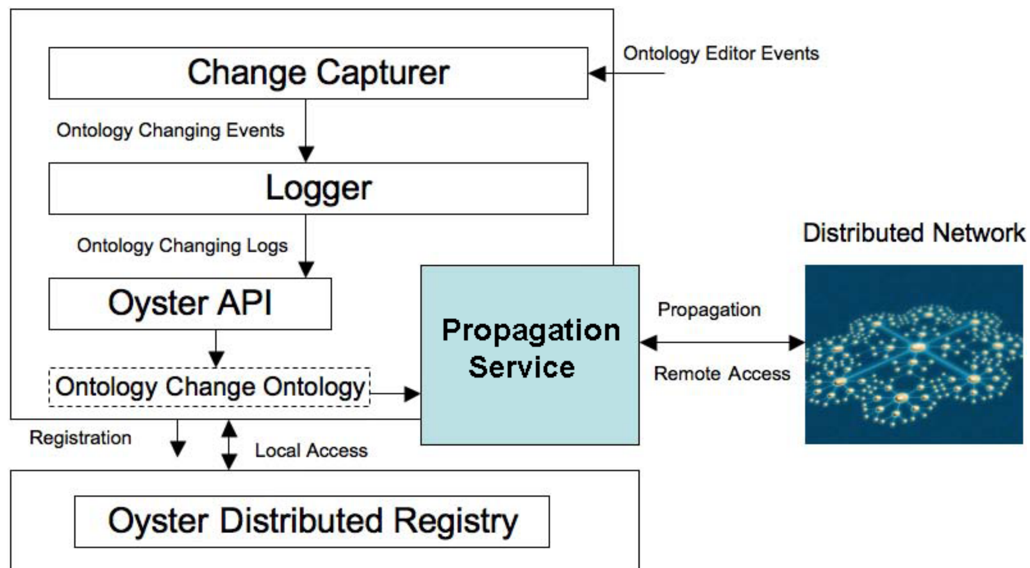

Figure 4.3: The architecture of Change Capturing plug-in in the distributed networking scenario

this architecture.

Figure 4.3 doesn't include the correspondence between this Change Capturing plugin with plugins that are going to be developed in the feature. However, it includes the relationship with Change Propagation service that is introduced later in this chapter.

4.1.3 Versioning component

The versioning component is in charge of the ontology change management which includes keeping track of the ontology changes and the ability to identify and maintain different variants of one ontology and their dependencies with support to undo/redo operations. The versioning component is implemented as part of Oyster.

To keep the track of the ontology changes we generate a log that maintains the history (and order) of applied changes as a sequence of individuals of our proposed change ontology by registering ontology changes in Oyster. The information about the precise changes performed is used to compute the difference between variants (see ontology comparison component), or to to undo/redo operations.

Registering Changes

In the first prototype of our implementations we rely in two APIs for the management of changes in Oyster (i.e. in the future we will have also the option to use the web service):

OMV API The classes defined by the OMV Ontology (and extensions) have been converted into java objects to provide a standard OMV API that can be used by java developers for the management of OMV objects (e.g. get/set properties)[1].

Oyster API Oyster API provides a set of methods for e.g. connection management, register metadata, update metadata, remove metadata and query metadata. Methods in the API use OMV objects defined in

[1]The OMV API is freely available at `http://www.ontoware.org/projects/oyster2`

the OMV API for an easy development of applications.

Hence, to store an ontology change in Oyster: first, we use the OMV API to create a java object of type OMVChange (e.g. entityChange). Then we use the method *register* from Oyster API to send it to Oyster. The API translates the java object into an instance of the change ontology and store it into the local repository.

Identifying ontologies

For the identification of ontologies, Oyster follows the approach proposed in [HP05] to identify an ontology metadata entry: it uses a tripartite identifier consisting of the Ontology URI plus an optional version information plus the location of the particular ontology implementation annotated.

4.1.4 Change Propagation Service

For the propagation of the ontology changes to the ontology metadata, we are implementing the WS-Notification mechanism introduced in 3.4. We defined a pre-defined topic associated to ontology changes to which one(or more) of the nodes in the distributed network subscribe. Using a pull approach, nodes request the required changes when new notifications become available. Each notified node in Oyster will trigger the procedure to re-validate the metadata annotation (i.e. validate if the content of the current metadata is consistent with the changes in the ontology or if it can be automatically updated accordingly).

Then, for the synchronization of the ontology metadata annotations in the network, each instance of Oyster periodically contacts other nodes in the network to exchange updated information. For this task, each metadata annotation in Oyster has the information about the last modification time of the associated ontology. Hence, during the the synchronization, as proposed in 3.4, every metadata annotation in the local repository is updated with the corresponding information from a remote node if the the local modification time of the metadata annotation is lower that then modification time in the remote node.

4.1.5 Ontology Comparison

One of the advantages of computing ontology comparison at different levels is that these levels can be implemented independently, and combined in an overall procedure as described in Section 3.6. Most of these steps are very simple to implement as they only require the use of set manipulations. However, it is important the keep the implementation of these set operations as efficient as possible.

In the current implementation, we provide methods for syntactic ontology comparisons within the OWL Tools plugin (`http://ontoware.org/projects/owltools`). The implementation performs a simple set difference of the axioms, but is additionally also able to cope with changes is namespaces (e.g. for the case where namespaces are used as a versioning scheme).

Comparisons at semantic level are more complicated and require the use of a reasoner. This function is at the basis of the implementation of the two levels of semantic comparison that are considered in our approach.

4.1.6 Conflict Resolution

As discussed in Section 3.5, we are considering two types of conflicts in the collaborative workflow: (1) logical conflicts in the form of inconsistencies and (2) conflicts due to concurrent editing of an ontology.

To deal with logical contradictions, we provide the RaDON plugin `http://ontoware.org/projects/radon/`. RaDON provides a set of techniques for dealing with inconsistencies and incoherence in ontologies. In particular, RaDON supports novel strategies and consistency models for distributed and networked environments.

For the first prototype we will not allow concurrent editing of the ontology, instead we assume a sequential workflow. For a later prototype, the conflict resolution for concurrent editing will be addressed in the

2006–2008 © Copyright lies with the respective authors and their institutions.

implementation of the NeOn Toolkit Collaboration Server. To realize collaborative ontology modeling two components must be developed:

- a server component that stores and manages ontologies and also handles remote clients that create or retrieve change events
- a NeOn Toolkit plug-in that represents the client-side implementation which communicates with the server component and acts as a proxy for the other components of the local ontology engineering environment.

Since the NeOn Toolkit is based on the KAON2 API for managing ontologies, the client-component is realized as an implementation of a Kaon2Connection.

This implementation accesses the specified collaboration server but locally appears like a normal datamodel implementation, since it implements the Kaon2Connection/OntologyManager interface. The server component consists of a normal KAON2 datamodel implementation, extended by means to accept remote clients, and to retrieve remote requests, queries and changeEvents from these clients. The basic task of the collaboration server is the marshaling and unmarshaling of objects that are passed between client and server.

The collaboration server realizes a synchronous, distributed datamodel. Due to the synchronous character, modeling conflicts can only occur due to network latency and when multiple modelers modify the same entity at the same time. Since the granularity of changes is very small, this is unlikely to happen. Nevertheless it cannot theoretically be ruled out.

The collaboration server cannot detect editing conflicts and the resulting states of the clients might be slightly different for a small fraction of the ontology. Though, a refresh of these parts synchronizes all clients with the central storage. One client might not see the changes he made to the model, since they have been over-ruled by parallel changes from another client. Nevertheless, the state of the ontology is always sound and immediate feedback alleviates for the fact that small conflicts between concurrent updates can potentially occur.

For simple changes on the instance level it is planned to improve the conflict resolution. With an extended API (not implemented yet) the client sends both the old state of an instance and the new state to the server. If in the meantime another client has changed the state of the instance, the server can detect that the first client has outdated assumptions about the instance state and can either merge the changes if there are no conflicts on the detail level or it can reject the changes and inform the client about the concurrent change.

Chapter 5

Conclusions and Outlook

5.1 Summary

In this deliverable, we first collected the requirements for designing the collaborative workflow for ontology editing tasks. Then, we introduced models and strategies for propagating networked ontologies that are defined in NeOn Project deliverables D1.1.1 and D1.1.2. We defined a change ontology to support capturing the changes of ontologies. We mainly focused on six aspects in modeling the processes of ontology change management:

- ontology change representation and capturing
- ontology versioning
- workflow model for collaborative ontology editing
- ontology change propagation models
- ontology conflict resolution
- ontology comparison

We identified our approach to implemented the above listed aspects. Finally, we developed several components as plug-ins in NeOn Toolkit to support the OWL Ontology Editor as the basis to support workflow of ontology editing. In Oyster, preliminary API support for the management of ontology changes described using our proposed change ontology and support for different versions of ontologies were implemented. Comparison components were also introduced as part of the provided components. We implemented the Ontology Change Capturing component and integrated it in the OWL Ontology Editor in NeOn Toolkit V1.0. During this deliverable we used the case study task T7.4 as test case to show the applicability of the research work and direct link among workpackages. Nevertheless our approach is applicable to any other scenario with similar requirements. Even more, the methods and corresponding components described in this deliverable can be used individually to provide support to simpler use cases or can be later on integrated with additional components to provide additional features. For instance, in a simpler scenario where is only necessary to support the tracking of changes, it would be enough to use the change capturing component within the OWL Ontology Editor and the Oyster system to store the ontology changes modelled using the change ontology.

5.2 Future Work

There are many challenges that will be addressed in the future work. First we need to finish the implementation of some of the methods proposed in this deliverable. We will further enhance and evaluate our approach and provide additional functionalities of collaborative ontology editing in the OWL Ontology Editor

2006–2008 © Copyright lies with the respective authors and their institutions.

of the NeOn Toolkit. We expect to have the components of the first prototype implemented by the end of the second year of NeOn (March 2008), and the whole infrastructure by the end of the third year (March 2009).

- We need to evaluate the proposed taxonomy of changes based on the feedback received on our first prototype of tools supporting ontology propagation models and strategies
- We will provide a more complete mapping between changes at the ontology element level with changes at the axiom level
- We will finish the implementation and evaluation of the model for the ontology change propagation in the near future
- Provide a formal model that represents the changes in metadata according to the changes in the ontology
- Evaluate the synchronization of the Oyster distributed registry
- We need to build GUIs for the visualisation of different ontology versions along with comparison components.
- We will upgrade our support to the incoming OWL-based ontology editor in NeOn Toolkit.
- The visualization of ontologies in the collaborative workflow with a clear difference in the state of each ontology element will be supported in the future.

Bibliography

[BG04] D. Brickley and R. V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. W3C Rec. 10 February 2004, 2004. available at `http://www.w3.org/TR/rdf-schema/`.

[BKKK87] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and implementation of schema evolution in object-oriented databases. *SIGMOD Rec.*, 16(3):311–322, 1987.

[BLFM98] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifiers (uri): Generic syntax, 1998.

[BR00] Keith H. Bennett and Vaclav Rajlich. Software maintenance and evolution: a roadmap. In *ICSE - Future of SE Track*, pages 73–87, 2000.

[CAM+07] Óscar Corcho, Pinar Alper, Paolo Missier, Sean Bechhofer, Carole A. Goble, and Wei Xing. Metadata management in s-ogsa. In Yong Shi, G. Dick van Albada, Jack Dongarra, and Peter M. A. Sloot, editors, *International Conference on Computational Science (2)*, volume 4488 of *Lecture Notes in Computer Science*, pages 712–719. Springer, 2007.

[CEM01] Pierre-Antoine Champin, Jérôme Euzenat, and Alain Mille. Why urls are good uris, and why they are not, 2001.

[D05́] Alicia Díaz. *Supporting Divergences in Knowledge Sharing Communities*. PhD thesis, Laboratoire Lorrain de Recherche en Informatique et ses Applications UMR 7503, 2005.

[DKP+01] Pavan Deolasee, Amol Katkar, Ankur Panchbudhe, Krithi Ramamritham, and Prashant J. Shenoy. Adaptive push-pull: disseminating dynamic web data. In *World Wide Web*, pages 265–274, 2001.

[dMLM06] A. de Moor, P. De Leenheer, and R.A. Meersman. DOGMA-MESS: A meaning evolution support system for interorganizational ontology engineering. In *Proc. of the 14th International Conference on Conceptual Structures, (ICCS 2006), Aalborg, Denmark*, Lecture Notes in Computer Science. Springer-Verlag, 2006.

[GAGW06] Jose Manuel Gómez, Carlos Buil Aranda, Oscar Munoz Garcí, and Walter Waterfeld. D6.1.2 context languages - state of the art. Technical Report D6.1.1, iSOCO, August 2006.

[GLP+07] Aldo Gangemi, Jos Lehmann, Valentina Presutti, Malvina Nissim, and Carola Catenacci. C-odo: an owl meta-model for collaborative ontology design. In Harith Alani, Natasha Noy, Gerd Stumme, Peter Mika, York Sure, and Denny Vrandecic, editors, *Workshop on Social and Collaborative Construction of Structured Knowledge (CKC 2007) at WWW 2007*, Banff, Canada, 2007.

[HBP+07] Peter Haase, Saartje Brockmans, Raul Palma, Jérôme Euzenat, and Mathieu d'Aquin. Updated version of the networked ontology model. Technical Report D1.1.2, University of Karlsruhe, AUG 2007.

2006–2008 © Copyright lies with the respective authors and their institutions.

[HH00] Jeff Heflin and James A. Hendler. Dynamic ontologies on the web. In *AAAI/IAAI*, pages 443–449, 2000.

[HP05] J. Hartmann and R. Palma. OMV - Ontology Metadata Vocabulary for the Semantic Web, 2005. v. 1.0, available at `http://omv.ontoware.org/`.

[HRW+06] Peter Haase, Sebastian Rudolph, Yimin Wang, Saartje Brockmans, Raul Palma, Jéróme Euzenat, and Mathieu d'Aquin. D1.1.1 networked ontology model. Technical Report D1.1.1, Universität Karlsruhe, NOV 2006.

[HSV04] Peter Haase, York Sure, and Denny Vrandecic. Ontology Management and Evolution: Survey, Methods and Prototypes. Technical report, Institute AIFB, University of Karlsruhe, December 2004.

[KF01] M. Klein and D. Fensel. Ontology versioning for the semantic web, 2001.

[KFK+02] Michel Klein, Dieter Fensel, Atanas Kiryakov, Natasha F. Noy, and Heiner Stuckenschmidt. Versioning of distributed ontologies. Technical report, Vrije Universiteit Amsterdam, December 2002. Internet: http://wonderweb.semanticweb.org/deliverables/documents/D20.pdf.

[Kle02] M. Klein. Ontology versioning and change detection on the web, 2002.

[Kle04] Michel C. A. Klein. *Change Management for Distributed Ontologies*. PhD thesis, Vrije Universiteit, Amsterdam), 2004.

[KN03] M. Klein and N. Noy. A component-based framework for ontology evolution. In *Proceedings of the IJCAI'03 Workshop: Ontologies and Distributed Systems*, Acapulco, Mexico, 2003.

[LAS06] Yaozhong Liang, Harith Alani, and Nigel Shadbolt. Enabling active ontology change management within semantic web-based applications. mini-thesis: Phd upgrade report, 2006.

[Ler00] Barbara Staudt Lerner. A model for compound type changes encountered in schema evolution. *ACM Transactions on Database Systems*, 25(1):83–127, 2000.

[LH90] Barbara Staudt Lerner and A. Nico Habermann. Beyond schema evolution to database reorganization. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 67–76, New York, NY, USA, 1990. ACM Press.

[LM07] Pieter De Leenheer and Tom Mens. *Ontology Management. Semantic Web, Semantic Web Services, and Business Applications*, chapter ONTOLOGY EVOLUTION. State-of-the-art and Future Directions. check, 2007.

[MAC+06] P. Missier, P. Alper, O. Corcho, I. Kotsiopoulos, I. Dunlop, W. Xing, S. Bechhofer, and C. Goble. Managing semantic grid metadata in s-ogsa. In *Cracow Grid Workshop 2006*, Cracow, Poland, 2006.

[MGKS+07] Muñoz-García, Soonho Kim, Marta Iglesias Sucasas, Caterina Caracciolo, Andrew Bagdanov, Yimin Wang, Peter Haase, Maria del Carmen, Suarez-Figueroa, and Asuncion Gomez-Perez. D7.4.1 software architecture for managing the fisheries ontologies lifecycle. Technical Report D7.4.1, Universidad Politecnica de Madrid, OCT 2007.

[NK03] Natalya Fridman Noy and Michel C. A. Klein. Tracking complex changes during ontology evolution. In *ISWC-2003 Poster Proceedings*, Sanibel Island, Florida, 2003. `http://www.stanford.edu/~natalya/papers/trackingChangesPoster.pdf`.

[NKKM04] Natalya Fridman Noy, Sandhya Kunnatur, Michel C. A. Klein, and Mark A. Musen. Tracking changes during ontology evolution. In *International Semantic Web Conference*, pages 259–273, 2004.

[NM02] Natalya F. Noy and Mark A. Musen. Promptdiff: A fixed-point algorithm for comparing ontology versions. In *National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence*, pages 744–750, Edmonton, Alberta, Canada, July 2002.

[NR89] G. T. Nguyen and D. Rieu. Schema evolution in object-oriented database systems. *Data Knowl. Eng.*, 4(1):43–67, 1989.

[OAS06] OASIS. Web Services Base Notification 1.3). Technical report, OASIS, October 2006.

[OK02] Damyan Ognyanov and Atanas Kiryakov. Tracking changes in rdf(s) repositories. In *EKAW '02: Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web*, pages 373–378, London, UK, 2002. Springer-Verlag.

[Oli00] D. Oliver. *Change Management and Synchronization of Local and Shared Versions of a Controlled Vocabulary*. PhD thesis, Stanford University, 2000.

[Pin04] S. Pinto. Ontoedit empowering swap: a case study in supporting distributed, loosely-controlled and evolving engineering of ontologies (diligent, 2004.

[PK97] A. Pons and R. K. Keller. Schema evolution in object databases by catalogs. In *IDEAS '97: Proceedings of the 1997 International Symposium on Database Engineering & Applications*, page 368, Washington, DC, USA, 1997. IEEE Computer Society.

[PÖ97] Randel J. Peters and M. Tamer Özsu. An axiomatic model of dynamic schema evolution in objectbase systems. *ACM Transactions on Database Systems*, 22(1):75–114, 1997.

[PS87] D.J. Penney and Jacob Stein. Class Modification in the GemStone Object-Oriented DBMS. In *Proceedings of the 2nd ACM Conference on Object-Oriented Programming Systems, Langages and Applications*, pages 111–117, Orlando, Florida, October 1987.

[QHJ07] Guilin Qi, Peter Haase, and Qiu Ji. D1.2.1 consistency models for networked ontologies. Technical Report D1.2.1, Universität Karlsruhe, FEB 2007.

[Rod95] John F. Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383–393, 1995.

[RR97] Young-Gook Ra and Elke A. Rundensteiner. A transparent schema-evolution system based on object-oriented view technology. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):600–624, 1997.

[SBF98] Rudi Studer, V. Richard Benjamins, and Dieter Fensel. Knowledge engineering: Principles and methods. *Data Knowledge Engineering*, 25(1-2):161–197, 1998.

[Sto04] Ljiljana Stojanovic. *Methods and Tools for Ontology Evolution*. PhD thesis, University of Karlsruhe (TH), Germany, August 2004.

[TN07] Tania Tudorache and Natasha Noy. Collaborative protege. In *Workshop on Social and Collaborative Construction of Structured Knowledge (CKC 2007) at WWW 2007*, Banff, Canada, 2007.

[Völ06] Max Völkel. D2.3.3.v2 SemVersion: Versioning RDF and Ontologies. Technical report, University of Karlsruhe, January 2006.

2006–2008 © Copyright lies with the respective authors and their institutions.