D3.1.1 Ontology Management and Evolution – Survey, Methods and Prototypes

Peter Haase, York Sure and Denny Vrandečić
(Institute AIFB, University of Karlsruhe)

Abstract.
EU-IST Integrated Project (IP) IST-2003-506826 SEKT
Deliverable D3.1.1 (WP1.1)
We present a survey of related work on ontology management and evolution. We then describe methods for evolution of OWL ontologies and their partial implementation in the prototypes *dlpconvert* and *evOWLution*. The appendix contains a detailed user guide for various tools of the KAON tool suite which forms a basic ontology management infrastructure.

Keyword list: ontology management, ontology evolution

Document Id.	SEKT/2004/D3.1.1/v1.0
Project	SEKT EU-IST-2003-506826
Date	December 20, 2004
Distribution	public

SEKT Consortium

This document is part of a research project partially funded by the IST Programme of the Commission of the European Communities as project number IST-2003-506826.

British Telecommunications plc.
Orion 5/12, Adastral Park
Ipswich IP5 3RE
UK
Tel: +44 1473 609583, Fax: +44 1473 609832
Contact person: John Davies
E-mail: john.nj.davies@bt.com

Jozef Stefan Institute
Jamova 39
1000 Ljubljana
Slovenia
Tel: +386 1 4773 778, Fax: +386 1 4251 038
Contact person: Marko Grobelnik
E-mail: marko.grobelnik@ijs.si

University of Sheffield
Department of Computer Science
Regent Court, 211 Portobello St.
Sheffield S1 4DP
UK
Tel: +44 114 222 1891, Fax: +44 114 222 1810
Contact person: Hamish Cunningham
E-mail: hamish@dcs.shef.ac.uk

Intelligent Software Components S.A.
Pedro de Valdivia, 10
28006 Madrid
Spain
Tel: +34 913 349 797, Fax: +49 34 913 349 799
Contact person: Richard Benjamins
E-mail: rbenjamins@isoco.com

Ontoprise GmbH
Amalienbadstr. 36
76227 Karlsruhe
Germany
Tel: +49 721 50980912, Fax: +49 721 50980911
Contact person: Hans-Peter Schnurr
E-mail: schnurr@ontoprise.de

Vrije Universiteit Amsterdam (VUA)
Department of Computer Sciences
De Boelelaan 1081a
1081 HV Amsterdam
The Netherlands
Tel: +31 20 444 7731, Fax: +31 84 221 4294
Contact person: Frank van Harmelen
E-mail: frank.van.harmelen@cs.vu.nl

Empolis GmbH
Europaallee 10
67657 Kaiserslautern
Germany
Tel: +49 631 303 5540, Fax: +49 631 303 5507
Contact person: Ralph Traphöner
E-mail: ralph.traphoener@empolis.com

University of Karlsruhe, Institute AIFB
Englerstr. 28
D-76128 Karlsruhe
Germany
Tel: +49 721 608 6592, Fax: +49 721 608 6580
Contact person: York Sure
E-mail: sure@aifb.uni-karlsruhe.de

University of Innsbruck
Institute of Computer Science
Techikerstraße 13
6020 Innsbruck
Austria
Tel: +43 512 507 6475, Fax: +43 512 507 9872
Contact person: Jos de Bruijn
E-mail: jos.de-bruijn@deri.ie

Kea-pro GmbH
Tal
6464 Springen
Switzerland
Tel: +41 41 879 00, Fax: 41 41 879 00 13
Contact person: Tom Bösser
E-mail: tb@keapro.net

Sirma AI EAD, Ontotext Lab
135 Tsarigradsko Shose
Sofia 1784
Bulgaria
Tel: +359 2 9768 303, Fax: +359 2 9768 311
Contact person: Atanas Kiryakov
E-mail: naso@sirma.bg

Universitat Autonoma de Barcelona
Edifici B, Campus de la UAB
08193 Bellaterra (Cerdanyola del Vallès)
Barcelona
Spain
Tel: +34 93 581 22 35, Fax: +34 93 581 29 88
Contact person: Pompeu Casanovas Romeu
E-mail: pompeu.casanovas@uab.es

Executive Summary

This deliverable provides a comprehensive state of the art survey of the tools, processes, frameworks and methodologies available for Ontology Management and Ontology Evolution, also taking a look at related work in other fields of technology, especially databases and software engineering. There we identified numerous open research questions that led our way.

In the context of this project, the focus of the research is on developing a tool framework and methods for the evolution of OWL-based ontologies and their application for data integration scenarios in the presence of heterogeneous evolving data sources. We explored an approach to formalize the semantics of change for the OWL onology language (in particular for OWL DL and its sublanguages), embedded in a generic process for ontology evolution. Our formalization of the semantics of change allows to define arbitrary consistency conditions – grouped in structural, logical, and user-defined consistency – and to define resolution strategies that assign resolution functions to ensure these consistency conditions are satisfied as the ontology evolves. This flexibility allows to support various fragments of the OWL-DL language.

This lead us to the development of a software prototype, evOWLution, based in the KAON2 infrastructure which is currently being developed in the EU IST DIP[1] project. evOWLution implements the results of the described original research done in the SEKT project on the methods for a consistent evolution of OWL ontologies.

A second prototype developed within the framework of this deliverable is dlpconvert. It is a tool to convert an OWL encoded ontology, that lies within the DLP fragment, to another syntactic representation. The DLP fragment has certain computational advantages and actually covers by far most of the existing ontologies. This is a further step on realizing language independent modelling of knowledge bases, and thus increasing the number of possible tools.

Using the research done on ontology evolution, especially on consistency conditions and consistent evolution, we will gain a lot in developing and evolving ontologies that lie within a certain well defined fragment like the DLP fragment.

In the appendix we describe the KAON framework in greater detail, in order to allow partners and interested readers to work with KAON, the KAON API and numerous KAON-based tools and extension efficiently.

[1] see `http://dip.semanticweb.org`

Contents

Chapter 1

Introduction

1.1 The SEKT Big Picture

This report is part of the work performed in workpackage (WP) 3 on "Ontology and Metadata Management". As shown in Figure 1.1 this work belongs to the central part of the research and development WPs in SEKT. Quite naturally it is closely connected with Ontology Generation and Metadata Generation, in particular we will integrate parts of their technologies. We are focusing on how to manage ontologies (and related metadata) and their evolution over time. As part of WP3.1, we will provide a basic infrastructure for ontology management. We will extend this in WP3.2 and WP3.3 with functionalities for data-driven change discovery and usage tracking, i.e. with means to adapt ontologies according to underlying domain knowledge in form of documents on the one hand and the usage of ontologies in applications by users on the other hand. As part of WP7 Methodology we will closely collaborate with the case study partners to apply our technologies within the case studies (see e.g. [EGH+04b, EGH+04a, ST04] and following ones).

1.2 Ontologies

Initially introduced by Aristotle, ontologies recently have become a topic of interest in computer science. Ontologies provide a shared understanding of a domain of interest to support communication among human and computer agents, typically being represented in a machine-processable representation language. Thus, ontologies are seen as key enablers for the Semantic Web [BLHL01]. Standards for ontology languages include the layered W3C standards XML/S, RDF/S and OWL. There exist numerous scientific and commercial tools for creating and maintaining ontologies (see Chapter 2.5), which have been used to build applications based on ontologies, including the areas of knowledge management, engineering disciplines, medicine or bio-informatics.

However, those pieces of knowledge so far have been treated mainly as being static. In reality they evolve over time, sometimes they even have a highly dynamic nature (see e.g. Peer-to-Peer scenarios such as Bibster [HEHS04]). Domain changes, adaptations to

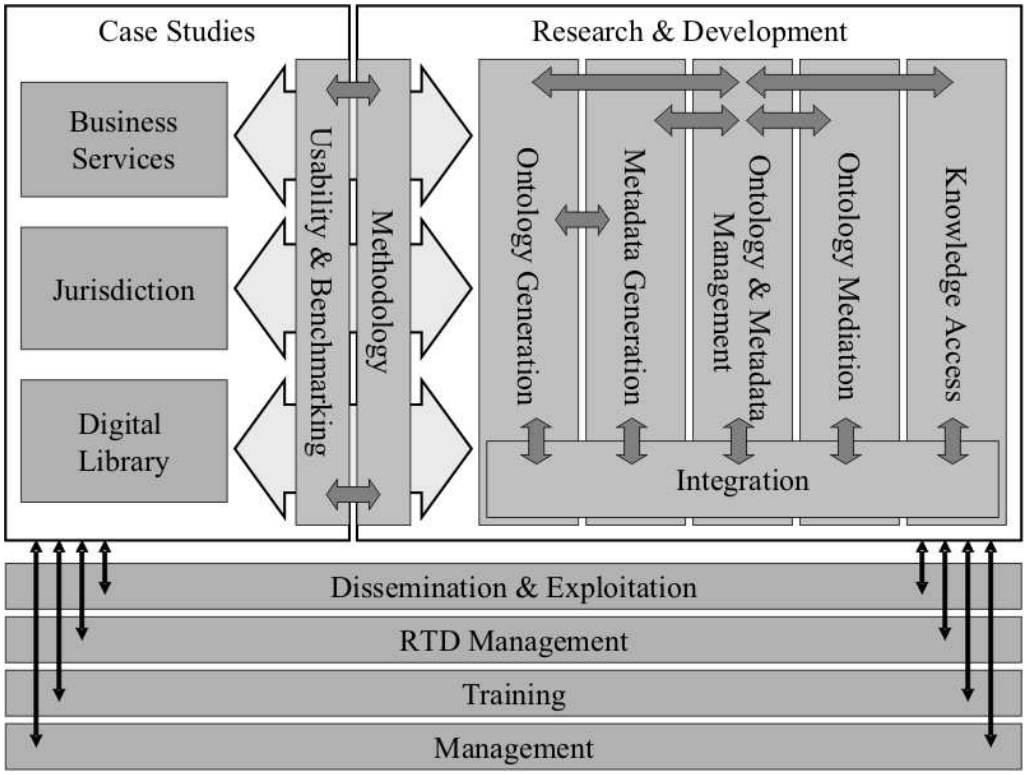

Figure 1.1: The SEKT Big Picture

different tasks, or changes in the conceptualization require modifications of the ontology.

1.3 Definition

We will now define some terms which are most relevant for this document.

According to [Sto04b], "Ontology Evolution is the timely adaptation of an ontology to the arisen changes and the consistent propagation of these changes to dependent artefacts." The author describes ontology evolution as a process, as changes in the ontology can cause inconsistencies in other parts of the ontology, as well as in the dependent artefacts. The ontology evolution process encompasses the set of activities, both technical and managerial, ensuring that the ontology continues to meet organizational objectives and users needs in an efficient and effective way.

In [SMMS02a] the authors identify a possible six-phase evolution process, the phases being: (1) change capturing, (2) change representation, (3) semantics of change, (4) change implementation, (5) change propagation, and (6) change validation. In the following, we will use this evolution process as the basis for an analysis of state-of-the-art technology.

Further, it is important to distinguish between the management, modification, evo-

lution and versioning of ontologies. In this document we follow the terminology of [Sto04b], which has been adapted from the terminology from the database community [Rod95]:

- **Ontology management** is the whole set of methods and techniques necessary to efficiently use multiple variants of ontologies from possibly different sources for different tasks. Therefore, an ontology management system should be a framework for creating, modifying, versioning, querying, and storing ontologies. It should allow an application to work with an ontology without worrying about how the ontology is stored and accessed, how queries are processed, etc.;
- **Ontology modification** is accommodated when an ontology management system allows changes to the ontology that is in use, without considering the consistency;
- **Ontology evolution** is accommodated when an ontology management system facilitates the modification of an ontology by preserving its consistency;
- **Ontology versioning** is accommodated when an ontology management system allows handling of ontology changes by creating and managing different versions of it.

1.4 Overview

This deliverable is organized as follows. We firstly present our results of a review of related work in the area in Chapter 2. We have in particular analyzed the current state of the art of ontology evolution on the research side on the one hand and existing tools on the other hand. We have identified the work relevant for the SEKT project, which in particular requires support for the evolution of OWL ontologies.

In Chapter 3 we present methods for the evolution of OWL ontologies that focus on the semantics of change operations, allowing to maintain consistency as the ontologies change.

We finally present three prototypes for ontology management and evolution: The KAON tool suite for ontology management and evolution, dlpconvert – a tool for converting OWL ontologies to Datalog, and OWL Evolution – a software component to support the consistent evolution of OWL ontologies.

Chapter 2

Ontology Evolution – Survey

This chapter is structured as follows. First we will present an overview of the relevant research area. In Section 2.5 the most relevant existing tools for ontology evolution and versioning are being described. Section 2.6 presents related research according to the evolution process described in Section 2.1, but also covers ontology versioning and evolution/versioning in database systems. Finally, we conclude in Section 2.7.

2.1 Ontology Evolution Process and Frameworks

In [SMMS02a] the authors identify a possible six-phase evolution process, the phases being: (1) change capturing, (2) change representation, (3) semantics of change, (4) change implementation, (5) change propagation, and (6) change validation. In the following, we will use this evolution process as the basis for an analysis of state-of-the-art technology.

Change Capturing The process of ontology evolution starts with capturing changes either from explicit requirements or from the result of **change discovery** methods, which induce changes from existing data. Explicit requirements are generated, for example, by ontology engineers who want to adapt the ontology to new requirements or by the end-users who provide the explicit feedback about the usability of ontology entities. The changes resulting from this kind of requirements are called *top-down* changes. Implicit requirements leading to so-called *bottom-up* changes are reflected in the behaviour of the system and can be discovered only through the analysis of this behaviour. [Sto04b] defines three types of change discovery: structure-driven, usage-driven and data-driven. Whereas structure-driven changes can be deduced from the ontology structure itself, usage-driven changes result from the usage patterns created over a period time. Data-driven changes are generated by modifications to the underlying dataset, such as text documents or a database, representing the knowledge modelled by an ontology.

Change Representation To resolve changes, they have to be identified and represented in a suitable format. That means, the change representation needs to be defined for a

given ontology model. Changes can be represented on various levels of granularity, e.g. as elementary or complex changes. A common practice is to provide a taxonomy or ontology of changes for a given ontology model.

Semantics of Change The semantics of change refers to the effects of the change on the ontology itself, and, in particular the checking and maintenance of the ontology consistency after the change application. The meaning of consistency depends heavily on the underlying ontology model. It can for example be defined using a set of constraints, as in the KAON ontology model, or it can be given a model-theoretic definition.

Change Propagation Ontologies often reuse and extend other ontologies. Therefore, an ontology update might also corrupt ontologies depending on the modified ontology (through inclusion, mapping integration, etc.) and consequently, all the artefacts based on these ontologies. The task of the change propagation phase of the ontology evolution process is to ensure consistency of *dependent artefacts* after an ontology update has been performed. These artefacts may include dependent ontologies, instances, as well as application programs running against the ontology.

Change Implementation The role of the change implementation phase of the ontology evolution process is (i) to inform an ontology engineer about all consequences of a change request, (ii) to apply all the (required and derived) changes and (iii) to keep track about performed changes.

Change Validation There are numerous circumstances where it may be desired to reverse the effects of the ontology evolution, to name just a few:

- The ontology engineer may fail to understand the actual effect of the change and approve the change that should not be performed;
- It may be desired to change the ontology for experimental purposes;
- When working on an ontology collaboratively, different ontology engineers may have different ideas about how the ontology should be changed.

It is the task of the change validation phase to recover from these situations. Change validation enables justification of performed changes and undoing them at user's request. Consequently, the usability of the ontology evolution system is increased.

2.2 Ontology Versioning

Ontology versioning is a stronger variant of handling changes to ontologies: While ontology evolution is concerned about the ability to change an ontology without losing data and by maintaining consistency, ontology versioning allows to access the data through

different variants of the ontology. In addition to managing the individual variants of the ontology themselves, it is also important to manage the derivation relations between the variants. These derivation relations then allow to define the notions of compatibility between versions, mapping relations between versions, as well as transformations of data corresponding to the various versions.

2.3 Evolution and Versioning in Database Systems

The problem of schema evolution has been extensively studied especially in the context of object-oriented databases. Dynamic schema evolution in databases is defined as managing schema changes in a timely manner without loss of existing data while the database system continues to be operational and without significantly impacting day-to-day operations of the database. Particular problems addressed are cascading changes (changes required to other parts of the schema as a result of a change), ensuring consistency of the schema, and propagation of the changes to the corresponding database.

Although there are significant differences between schema evolution and ontology evolution [NK03], many of the methods and technologies developed for schema evolution can be applied or adapted to ontology evolution.

2.4 Evolution and Versioning for Other Paradigms

The problem of evolution and versioning is also present in other application areas of information systems.

For example, Concurrent Versions Systems (CVS) allow the concurrent update to files while maintaining the version history of those files as well as detecting and resolving conflicts in updates to the files.

Another application are is the maintenance of knowledge-base systems and belief revision, where a knowledge base (e.g. the believes of an agent) needs to be update to incorporate new information while maintaining consistency of the knowledge base.

2.5 Existing Tools

There are only a few existing tools that support the complete ontology evolution process. We therefore provide an overview of tools that only support specific aspects in the ontology evolution process, too. Further, we also cover non-commercial tools, i.e. tools that have come out of research projects. Before we illustrate the tools which already exist for ontology evolution and versioning we begin with one of the most prominent systems for versioning, the CVS, and discuss its drawbacks for the usage with ontologies.

2.5.1 Concurrent Version System – CVS

The very popular concurrent version system (CVS[1]) initially was a collection of scripts to simplify the handling of the revision control system (RCS). RCS operates in a file-centric way by using a "lock-modify-unlock"-process. CVS still relies on the RCS file format for storing versioning information, but it extends RCS e.g. by supporting network capabilities, by separating local copies and central repositories and by allowing parallel access of multiple users. The CVS basic version control functionality maintains a history of all changes made to directory trees, *i.e.* a hierarchy of file folders which might contain arbitrary file formats (often text files). The complete functionalities are described in the "official" manual for CVS [C+03].

However, CVS works on the syntactical level, not on the conceptual. *I.e.*, it is not capable of versioning objects and in particular not capable of versioning ontological entities and their complex structure. The underlying `diff` operation is capable of showing the syntactical differences between two files (based on the differences of text lines). Therefore, it is suitable to act as a very primitive versioning system *e.g.* for RDF/S or OWL *files*.

In a nutshell, standard `diff`, and thus CVS, compares file versions at line-level or at character-level, highlighting the lines that textually differ in two versions. Actually needed is a comparison of ontologies at a structural level, showing which definitions of ontological concepts or relationships have changed.

Nevertheless, CVS has already been used for inspiration for the ontology versioning system OntoView (see [Kle04] and next Subsection 2.5.4).

2.5.2 Ontology Editors

Ontology editors are tools that allow users to visually manipulate ontologies. In this subsection, we evaluate ontology editors in terms of the requirements for the ontology evolution. We select three ontology editors that are most frequently used in the Semantic Web community, a more complete overview can be found in [GPAFL+02].

- Protégé[2] (cf. [NFM00]) is a graphical and interactive ontology-design and knowledge-acquisition environment that is being developed by the Stanford Medical Informatics group (SMI) at Stanford University. Its knowledge model is compatible with OKBC (cf. [CFF+98]). Its component-based architecture enables system builders to add new functionality by creating appropriate plug-ins. The Protégé-OWL plug-in extends the Protégé platform into an ontology editor for the OWL;

- OntoEdit[3] (cf. [SAS03, SEA+02]) is an ontology engineering environment supporting the development and maintenance of ontologies by graphical means. OntoEdit is built on top of a powerful internal ontology model. This paradigm sup-

[1] Available freely for download at `http://www.cvshome.org/`
[2] `http://protege.stanford.edu/`
[3] `http://www.ontoprise.de/com/co_produ_tool3.htm`

ports representation-language neutral modelling as much as possible for concepts, relations and axioms. Several graphical views onto the structures contained in the ontology support modelling the different phases of the ontology engineering cycle. It has an interface to Ontobroker, an F-Logic Inference Engine;

- OilED[4] (cf. [BHGS01]) has been developed by the University of Manchester. It is a simple freeware ontology editor, which allows the user to build ontologies using OIL and OWL, and it is not intended as a full ontology development environment. Consistency checking and automatic classification of the ontologies written with it can be performed using the FaCT reasoner.

All editors allow modifications to ontologies in terms of elementary ontology changes. Even though composite changes allow an ontology engineer to update an ontology without having to find the right sequence of elementary modifications, most of the existing ontology editors do not include composite changes. Only OntoEdit provides support for some composite changes (e.g. copy).

Most of the existing systems for the ontology development provide only one possibility for realising a change, and this is usually the simplest one. For example, the deletion of a concept always causes the deletion of all of its subconcepts. It means that users are not able to control the way the changes are performed. Consequently, customisation is not supported at all.

Moreover, the users do not obtain explanations why a particular change is necessary (transparency). In OntoEdit, the user only obtains the information about numbers of induced changes but without providing more details. None of the existing editors warns ontology engineers about changes in the included ontologies.

Furthermore, there is no possibility to undo effects of changes (reversibility). Both, Protégé and OntoEdit, have an Edit menu with the Undo/Redo options. However, the performed changes are kept in the memory so that they are lost when the ontology/editor is closed.

Regarding auditing and the logging of changes, OilEd provides an activity log. However, it records connections to the reasoner, not all ontology modifications. Protégé also has the command history option but in the version we were dealing with it was useless since it was disabled.

2.5.3 Ontology Evolution in KAON

KAON is an open-source ontology management infrastructure targeted for semantics-driven business applications. It provides a comprehensive implementation allowing easy ontology management and application. Important focus of KAON is on integrating traditional technologies for ontology management and application with those used typically in business applications. A more detailed technical description of the KAON components can be found in [GSV04].

[4] `http://oiled.man.ac.uk`

Roughly, KAON components can be divided into three layers:

- the Applications and Services Layer realizes UI applications and provides interfaces to non-human agents. Among many applications realized, OntoMat-SOEP provides ontology and metadata engineering capabilities. It realizes many requirements related to ontology evolution and is described next in more detail.
- KAON API as part of the Middleware Layer is the focal point of KAON architecture since it realizes the model6 of ontology based applications. The bulk of requirements related to ontology evolution is realized in this layer and is described in the next subsection.
- Data and Remote Services Layer provides data storage facilities. This layer also realises concurrency and transactional atomicity of updates. Further elaboration of this layer is out of scope for this deliverable.

The focal point of the KAON architecture is its ontology API (KAON API), consisting of a set of interfaces for access to ontology entities. For example, there are the interfaces Concept, Property and Instance, which contain methods for accessing ontology concepts, properties and instances, respectively.

The API incorporates important elements required for ontology management and evolution:

- Evolution logging is responsible for keeping track of the ontology changes in an evolution log in order to be able to reverse them at the user s request. Further, the evolution log is also used by the distributed ontology evolution;
- Change reversibility enables undoing and redoing changes made in an ontology. Consequently, changes can be executed in reverse order thus forcing the ontology to return to the conditions prior to the change execution;
- Evolution strategy is responsible for ensuring that all changes applied to the ontology leave the ontology in a consistent state and for preventing illegal changes. Also, the evolution strategy allows the user to customise the evolution process;
- Evolution graph enables ontology engineers to enhance a set of changes with their own changes and to resolve them;
- Ontology inclusion facilities, together with the dependent evolution, are responsible for managing multiple ontologies within one node;
- Ontology replication facilities, together with the distributed evolution, are responsible for enabling the reuse and the management of distributed ontologies;
- Change discovery includes the means for the discovery of problems in an ontology and for making recommendation for their resolution;
- Usage logging is responsible for keeping track of the end-users interactions with ontology-based applications in order to adapt ontologies to the users needs.

Ontology Evolution in KAON API

Before the ontology evolution process is started, a particular evolution strategy must be configured. Changes to the ontology are performed by assembling elementary and composite changes into a sequence. However, before the ontology is actually updated, this sequence is passed to the present evolution strategy in the semantics of change phase, resulting in an extended sequence of changes. To ensure atomicity of updates, either all or no change from the extended sequence of changes should succeed, so validity of change sequence is checked before any updates are actually performed.

Transparency is realized by presenting the extended sequence of changed to the user for approval. To further aid the understanding of why some changes are performed, the evolution strategy may group related elementary actions and provide explanations why particular change is necessary, thus greatly increasing the chances that all side-effects of changes will be properly understood. After changes are reviewed by the user, they are passed to the ontology and executed, performing steps from the change implementation phase.

It is obvious that for each elementary change there is exactly one inverse change that, when applied, reverses the effect of the original change. With such infrastructure in place, it is not hard to realize the reversibility: to reverse the effect of some extended sequence of changes, a new sequence of inverse changes in reverse order needs to be created and applied. An evolution log associates additional information with each change. Effectively, the log is treated as an instance of a special evolution ontology (cf. Subsection 2.6.1) consisting of concepts for each change, making it is easy to add meta-information to log entries. The structure of the log may be easily customized by editing the evolution ontology. Further, available services for persisting ontology data may be used to persist the log, removing the need to devise yet another type of persistent storage. Evolution logging and reversibility services are provided as special services of the KAON API, allowing different applications to reuse these powerful features. E.g., actions performed in one application may be easily reverted in another.

Ontology Evolution in the OI-modeller

As mentioned in the previous subsection, the ontology evolution is primarily realised through the KAON API. However, UI applications provide human-computer interaction for the evolution, whose primary role is to present the change information in an orderly way, allowing easy spotting of potential problems. Also, any application that changes the ontology must realise the reversibility requirement in its user interface as well. Part of the KAON framework is the OI-modeller, an ontology and metadata engineering tool. It is an end-user application that realises a graph-based user interface for single, dependent and distributed ontology development. OI-modeller supports ontology evolution at the user level.

Currently evolution requirements are realised within the OI-modeller, as follows:

- An ontology engineer may set up the desired evolution strategy. It can be seen that

an evolution strategy consisting of several resolution points. For each resolution point the ontology engineer must choose appropriate elementary evolution strategy;

- Before changes are performed, the system computes the set of additional changes that must be applied. The impact of a change is reported to the ontology engineer. Presentation of changes follows the progressive disclosure principle: related changes are grouped together and organised in a tree-like form. The ontology engineer initially sees only the general description of changes. If she is interested in details, she can expand the tree and view complete information. Only when the ontology engineer agrees the changes will be applied to the ontology. The ontology engineer may cancel the operation before it is actually performed.

- An unlimited undo-redo function is provided. Although this function is by large the responsibility of the KAON API, the user interface is responsible for restoring the visual context after an undo operation. For example, if a concept in hierarchy was selected and then deleted, when an operation is undone, the same concept must be selected. If the hierarchy was scrolled in the meanwhile, the original scroll position must be restored. These features are necessary for the ontology engineer to quickly recognise a familiar state and proceed with her work. If not done properly, although an action is undone, the ontology engineer may not realise this and may mistakenly request another undo operation.

A detailed description about the OI-modeller and the KAON API including the support for evolution can be found in [GSV04].

2.5.4 OntoView

In [KFKO02] the authors describe the design of a web-based system that helps users to manage changes in ontologies. The system helps to keep different versions of web-based ontologies interoperable, by maintaining not only the transformations between ontologies, but also the conceptual relations between concepts in different versions. OntoView is inspired by the Concurrent Versioning System CVS (see previous subsection), which is used in software development to allow collaborative development of source code. The first implementation is also based on CVS and its web-interface CVSWeb.

The versioning system of OntoView provides the following functionalities:

1. Reading changes and ontologies
2. Identification of ontologies
3. Comparing ontologies at a conceptual level
4. Analyzing effects of changes, e.g. by checking consistency
5. Exporting changes

2.5.5 OntoManager

OntoManager[SHG03] is a tool for guiding ontology managers through the modification of an ontology with respect to users' needs. It is based on the analysis of end-users' interactions with the ontology-based applications, which are tracked in an usage-log.

OntoManager has been designed to provide the methods and tools that support the ontology managers in managing and optimising the ontology according to the users' needs. The system incorporates mechanisms that assess how the ontology (and by extension the application) is performing based on different criteria, and then enable to take action to optimise it. One of the key tasks is to check how the ontology fulfils the perceived needs of the users. In that way, an in-depth view of the users' perspective on the ontology and the ontology-based application is obtained, since on the top of this ontology the application is going to be conducted. The technique that can be used to evaluate/estimate the user needs depends on the information source. By tracking user interactions with the application in a log file, it is possible to collect useful information that can be used to assess what the main interests of the users are. In this way, it is avoided to ask the users explicitly, since they tend to be reluctant to provide the feedback via filling questionnaires or forms.

Conceptually, the OntoManager consists of three modules:

- The Data Integration Module that aggregates, transforms and correlates the usage data;
- The Visualisation Module that makes the integrated usage data more useful for human beings by presenting the data in a comprehensible visual form;
- The Analysis Module that provides guidance for adapting and improving the ontology with respect to the users' needs.

With respect to ontology evolution, the "Analysis Module" is most important: In particular, there are two task of the Analysis module:

1. Ontology Evolution that provides guidance in the process of modifying the ontology and ensure the consistency of the updated ontology. This module keeps track of the changes and has the possibility to undo any action taken upon the ontology. The OntoManager imports the functionalities related to the ontology evolution process that are elaborated in [SMMS02a].
2. Crawling that completes newly created concepts with the most promising instances that can be found in an intranet or internet.

2.5.6 TextToOnto

TextToOnto [MV01] is a tool suite built upon KAON [KK04] in order to support the ontology engineering process by text mining techniques. Providing a collection of independent tools for both automatic and semi-automatic ontology extraction, it assists the user in creating and extending OIModels. Moreover, efficient support for ontology maintenance

is given by modules for ontology pruning and comparison. Further information can be found, e.g., in [GSV04]. In a nutshell, the current distribution of TextToOnto comprises the following tools:

- **TaxoBuilder** constructs concept hierarchies by applying either FCA [CST03] or a combination of Hearst-Patterns [Hea92], WordNet [Fel98] and various heuristics.
- **TermExtraction** creates new concepts from possibly relevant terms included in the corpus. The relevancy of a term is measured, for example, by means of its frequency or its TFIDF value.
- **InstanceExtraction** supports both semi-automatic and fully automatic learning of instances by using a combination of various patterns from [Hea92] and [HS98].
- **RelationExtraction** extracts conceptual relations in a semi-automatic way by applying one of two different approaches. While the first one is based on association rules [MS00], the second one applies a set of text patterns very similar to those defined by Hearst [Hea92].
- **RelationLearning**, in contrast to RelationExtraction supports both automatic *and* semi-automatic relation learning. Moreover not only a domain and a range, but also a name for each relation are extracted from the corpus. The approach being applied by RelationLearning is based on shallow text parsing which is used in order to detect typical co-occurrences of predicates and conceptual classes, derived from the ontology.
- **OntologyComparison** compares two ontologies with respect to lexical and conceptual aspects [MS02].
- **OntologyPruner** can be used to adapt an ontology to a domain-specific corpus. It prunes the ontology by suggesting concepts to be removed on the basis of their frequency within a given corpus.

Since TextToOnto does not keep any references between the ontology and the text documents it has been extracted from, it does not allow for mapping textual changes to the ontology. Therefore data-driven change discovery is not (yet) supported by current versions of TextToOnto. In the follow-up Text2Onto[5], a complete re-implementation and the official successor of TextToOnto, we will focus on overcoming this deficiency. A detailed requirements analysis, together with a first architecture proposal, will be released as part of Task 3.3.

[5] `http://ontoware.org/projects/text2onto/`

2.6 Past and current research

2.6.1 Ontology Evolution Process and Frameworks

We again use the evolution process presented in [SMMS02a] to structure this chapter. However, we also would like to mention other approaches to the evolution process and frameworks.

In [PK97] the authors define three major steps in the schema evolution process: 1) request specification, 2) identification of changes, and 3) implementation. The change capturing phase and the change validation phase are not covered. Regarding the core evolution process dealing with the consistency of a schema and its dependent artefacts, it does not treat the semantics of change problem and requires writing the transformations to update data if they must not get lost. Regarding the implementation of changes it allows to realise the evolution by view, by version or by the immediate update.

[KN03] introduces a component-based framework for ontology evolution. It is based on the different representations of ontology changes. The approach proposes a framework that integrates these representations. It covers the following tasks: (i) data transformation; (ii) ontology update; (iii) consistent reasoning; (iv) verification and approval; and (v) data access. The last task is related to ontology versioning.

Change Capturing

Please note that we will refine this subsection (including the following subsubsections) in future as part of our work in tasks T3.2 and T3.3.

Usage Driven Change Discovery Once ontologies reach certain levels of size and complexity, the decision about which parts are further relevant and which are outdated is a huge task for ontology engineers. Usage patterns of ontologies and their metadata allow for a detection of often or less often used parts, thus reflecting e.g. the interests of users in parts of ontologies. They can be e.g. derived from tracking querying and browsing behaviours of users during the application of ontologies.

Data Driven Change Discovery An ontology is often learnt or constructed in order to reflect the knowledge more or less implicitly given by a number of documents or a database. Therefore, any change to the underlying data set, such as a newly added document or a changed database entry, might require an update of the ontology. Data-driven Change Discovery can be defined as the task of deriving ontology changes from modifications to the knowledge representation it has been constructed from. Or, more formally:

Definition: Let *D* be a data set containing explicit or implicit knowledge, which is modified by a sequence *C1,...Cn* of change operations. And let the knowledge in D be explicitly represented by an ontology *O*. Then Data-driven Change Discovery can be defined as the task of adapting *O* in order to reflect the changes *C1,...Cn*.

A slightly different definition is given by [Sto04b], who defines Data-driven Change Discovery as the problem of deriving ontological changes from the ontology instances by applying techniques such as data-mining, Formal Concept Analysis (FCA, [GW99]) or various heuristics. For example, one possible heuristic might be: If no instance of a concept C uses any of the properties defined for C, but only properties inherited from the parent concept, C is not necessary. An implementation of this notion of Data-driven Change Discovery is included in the KAON tool suite [KK04, GSV04].

One very obvious difference between these two definitions is, that the latter always assumes an existing ontology, while the former can be applied to an empty ontology as well, but requires an evolving data set associated with this ontology. Moreover the following prerequisites must be fulfilled:

1. Knowledge about *general* relationships between data and ontology is required, since in case of newly added or modified data, additional knowledge has to be extracted and represented by the ontology.

2. Knowledge about *concrete* relationships between the data and ontology concepts, instances and relations is needed, because deleting or modifying information in the data set might have an impact on existing entities in the ontology.

If the ontology creation process is done manually, for example by a knowledge engineer, then both kinds of knowledge are represented somewhere in the mind of this knowledge engineer. In that case Data-driven Change Discovery also has to be done manually. On the other hand, if the process of creating the ontology is done semi- or fully automatically with the help of an ontology learning system such as TextToOnto [MV01], this knowledge has to be represented explicitly by the system. Of course, the first kind of knowledge is always given by the concrete implementation of ontology learning algorithms which are used. Therefore, in order to enable an existing ontology learning system to support Data-driven Change Discovery, it is necessary to make it store all available knowledge about concrete relationships between ontology entities and the data set. A more detailed requirements analysis will included in deliverable [HV04].

Change Representation

[Sto04b] derives a set of ontology changes for the KAON ontology model. The author specifies fine-grained changes that can be performed in the course of the ontology evolution. They are called elementary changes, since they cannot be decomposed into simpler changes. A taxonomy of elementary changes is derived as the cross product of the set of entities of the ontology model and the meta-change transformations *add* and *remove*.

The author also mentions that this level of change representation is not always appropriate and therefore introduces the notion of composite changes: A composite change is an ontology change that modifies (creates, removes or changes) one and only one level of neighbourhood of entities in the ontology. Examples for these composite changes

would be: “Pull concept up”, “Concept Copy”, “Split Concept”, etc. Further, the author introduces complex changes: A complex change is an ontology change that can be decomposed into any mix of at least two elementary and composite ontology changes.

In [KN03] Klein and Noy also state that information about change can be represented in many different ways. They describe different representations and propose a framework that integrates them. They show how different representations in the framework are related by describing some techniques and heuristics that supplement information in one representation with information from other representations. They further present an ontology of change operations, which is the kernel of the framework.

[Kle04] describes a set of changes for the OWL ontology language, based on an OWL meta-model. Unlike the previously mentioned set of KAON ontology changes, the author considers also *Modify*-operations in addition to *Delete* and *Add*-operations. Further, the taxonomy contains *Set* and *Unset*-operations for properties (e.g. to set transitivity). The author introduces an extensive terminology of change operations along two dimensions: *atomic* vs. *composite* and *simple* vs. *rich*:

	simple	rich
atomic	basic	complex
composite	complex	complex

Atomic operations are operations that cannot be subdivided into smaller operations, whereas *composite operations* provide a mechanism for grouping operations that constitute a logical entity. *Simple changes* can be detected by analysing the structure of the ontology only, whereas *rich changes* incorporate information about the implication of the operation on the logical model of the ontology, for their identification one thus needs to query the logical theory of the ontology (e.g. *ModifyDomainToSuperclass*). The authors also proposes a method for finding complex ontology changes. It is based on a set of rules and heuristics to generate a complex change from a set of basic changes.

As one can easily see, the terminology of [Kle04] is not consistent with the terminology introduced in [Sto04b]. Simply speaking, *elementary* and *complex* changes in [Sto04b] correspond to *atomic* and *composite* changes in [Kle04], respectively. In [Sto04b], there is no explicit corresponding distinction for *simple* vs. *rich* changes.

Both [Sto04b] and [Kle04] present an “ontology for ontology changes” for their respective ontology language and identified change operations.

There exist models for change representations for other ontology languages: A formal method for tracking changes in the RDF repository is proposed in [OK02]. The RDF statements are pieces of knowledge they operate on. The authors argue that during the ontology evolution, the RDF statements can be only deleted or added, but not changed. Higher levels of abstraction of ontology changes such as composite and complex ontology changes are not considered at all in that approach.

Semantics of Change

The semantics of change phase is the phase in the ontology evolution process that enables the resolution of ontology changes in a systematic manner by ensuring the consistency of

the ontology. In the following we provide an overview of various notions of consistency and approaches for the realization of the changes.

Consistency The goal of the semantics of change phase is to ensure that the application of ontology changes results in an ontology conforming to its consistency model. [Sto04b] defines consistency as: "A single ontology OI is defined to be consistent with the respect to its model if and only if it preserves the constraints defined for underlying ontology model." For example, in the KAON ontology model, the consistency of an ontology is defined using a set of constraints, called invariants. These invariants state for example that the concept hierarchy has to be a directed acyclic graph.

The OWL ontology language also defines structural constraints on valid ontologies for the various fragments. In particular, [BvHH$^+$] defines the following constraints for OWL DL:

- OWL DL requires a pairwise separation between classes, datatypes, datatype properties, object properties, annotation properties, ontology properties (i.e., the import and versioning stuff), individuals, data values and the built-in vocabulary. This means that, for example, a class cannot be at the same time an individual.
- In OWL DL the set of object properties and datatype properties are disjoint. This implies that the following four property characteristics: inverse of, inverse functional, symmetric, and transitive can never be specified for datatype properties
- OWL DL requires that no cardinality constraints (local nor global) can be placed on transitive properties or their inverses or any of their superproperties.
- Annotations are allowed only under certain conditions.
- Most RDF(S) vocabulary cannot be used within OWL DL. See the OWL Semantics and Abstract Syntax document [PSHH] for details.
- All axioms must be well-formed, with no missing or extra components, and must form a tree-like structure.
- Axioms (facts) about individual equality and difference must be about named individuals.

However, regarding the consistency of Description Logics, we can also provide a model-theoretic definition. [PSHH] defines consistency as follows:

> A collection of abstract OWL ontologies and axioms and facts is consistent with respect to datatype map D iff there is some interpretation I with respect to D such that I satisfies each ontology and axiom and fact in the collection.

Please note, that with this model-theoretic definition, a DL-ontology can only become inconsistent by adding axioms: If a set of axioms is satisfiable, it will still be satisfiable when any axiom is deleted. Other approaches, e.g. adding constraints, would require non-monotonic reasoning, which are beyond the scope of OWL ontologies.

We now provide a typical example of introducing model-theoretic inconsistencies in the evolution of a DL ontology. Suppose, we start out with a very simple ontology about animals:

$bird \sqsubseteq animal$ (All Birds are animals)
$bird \sqsubseteq fly$ (All birds can fly)
As the ontology evolves and more facts are added:
$dove \sqsubseteq bird$ (All doves are birds)
But as we add the following facts about penguins.
$penguin \sqsubseteq bird$ (All penguins are birds)
$penguin \sqsubseteq \neg fly$ (Penguins are not flying animals)
the ontology becomes inconsistent, as the concept penguin is unsatisfiable. Now, there may be many ways how to resolve this inconsistency. One possibility would be to reject either the change $penguin \sqsubseteq bird$ or $penguin \sqsubseteq \neg fly$ from the set of changes. However, both of these axioms are correct, and the user may not be happy with this decision. The most intuitive one may be to retract the axiom $bird \sqsubseteq fly$, but also this may not satisfy the user.

[Sto04b] describes and compares two approaches to verify ontology consistency:

1. a posteriori verification, where first the changes are executed, and then the updated ontology is checked whether it satisfies the consistency constraints

2. a priori verification, which defines a respective set of preconditions for each change. It must be proven that, for each change, the consistency will be maintained if (1) an ontology is consistent prior to an update and (2) the preconditions are satisfied.

Realization [SMMS02a] and [SMSS03] describe two approaches for the realization of the semantics of change, a procedural and a declarative one, respectively. In both these approaches, the KAON ontology model is assumed. The two approaches were adopted from the database community and followed to ensure the consistency in pursuing this semantics of change problem [FGM00]:

1. Procedural approach - this approach is based on the constraints, which define the consistency of a schema, and definite rules, which must be followed to maintain constraints satisfied after each change;

2. Declarative approach - this approach is based on the sound and complete set of axioms (provided with an inference mechanism) that formalises the dynamics of the evolution.

In [SMMS02a] (procedural approach) the authors focus on providing the user with capabilities to control and customize the realization of the semantics of change. They introduce the concept of an evolution strategy encapsulating policy for evolution with respect to user s requirements. To resolve a change, the evolution process needs to determine answers at many *resolution points* – branch points during change resolution where taking a different path will produce different results. Each possible answer at each resolution point is an *elementary evolution strategy*. A common policy consisting of a set of elementary evolution strategies, each giving an answer for one resolution point, is an *evolution strategy* and is used to customize the ontology evolution process. Thus, an evolution strategy unambiguously defines the way how elementary changes will be resolved. Typically a particular evolution strategy is chosen by the user at the start of the ontology evolution process.

In [SMSS03] (declarative approach) the authors present an approach to model ontology evolution as reconfiguration-design problem solving. The problem is reduced to a graph search where the nodes are evolving ontologies and the edges represent the changes that transform the source node into the target node. The search is guided by the constraints provided partially by user and partially by a set of rules defining ontology consistency. In this way they allow a user to specify an arbitrary request declaratively and ensure its resolving.

Change Implementation

We analyse the different roles of the change implementation phase: (i) to inform an ontology engineer about all consequences of a change request, (ii) to apply all the (required and derived) changes and (iii) to keep track about performed changes.

Change Notification In order to avoid performing undesired changes, a list of all implications to the ontology and dependent artefacts should be generated and presented to the ontology engineer, who should then be able to accept or abort these changes.

Change Application The application of a change should have transactional properties, i.e. (A) Atomicity, (C) Consistency, (I) Isolation, and (D) Durability. The approach of [Sto04b] realizes this requirement by the strict separation between the request specification and the change implementation. This allows to easily treat the set of change operations as one atomic transaction, as all the changes are applied at once.

Change Logging There are various ways to keep track of the performed changes. [Sto04b] proposes an *evolution log* based on an *evolution ontology* for the KAON ontology model. The evolution ontology covers the various types of changes, dependencies between changes (causal dependencies as well as ordering), as well as the decision making process.

Change Propagation

[MMS03a] presents an approach for evolution in the context of dependent and distributed ontologies. The authors define the notion of *Dependent Ontology Consistency*: A dependent ontology is consistent if the ontology itself and all its included ontologies, observed alone and independently of the ontologies in which they are reused, are single ontology consistent. *Push-based* and *Pull-based* approaches for the synchronization of dependent ontologies are compared. The authors follow a push-based approach for dependent ontologies on one node and present an algorithm for dependent ontology evolution.

Further, for the case of multiple ontologies on multiple nodes, the authors define *Replication Ontology Consistency* (An ontology is replication consistent if it is equivalent to its original and all its included ontologies (directly and indirectly) are replication consistent.) and here, for the synchronization between originals and replicas, the authors follow a pull-based approach.

Evolution in modular and distributed ontologies One particular challenge of change propagation arises in the context of modularization. [SK03] concentrates on the benefits of modular ontologies with respect to local containment of terminological reasoning. The authors define an architecture for modular ontologies that supports local reasoning by compiling implied subsumption relations. They further address the problem of guaranteeing the integrity of a modular ontology in the presence of local changes. They propose a strategy for analysing changes and guiding the process of updating compiled information.

The authors address the problem of guaranteeing the integrity of a modular ontology in the presence of a local change. They propose a strategy for analysing changes and guiding the process of updating compiled information. Ontology modules are connected by conjunctive queries. In order to make local reasoning independent of other modules, the authors use a knowledge compilation approach. The result of each mapping query is computed off-line and added as an axiom to the ontology module using that result. Once a query has been compiled, the correctness of reasoning can only be guaranteed as long as the concept hierarchy of the queried ontology module does not change. The authors propose a heuristic change detection mechanism that analyses changes with respect to their impact on the concept hierarchy. The set of changes they consider is not complete, as they focus only on changes regarding the concept hierarchy.

2.6.2 Ontology Versioning

In [HH00] the authors discuss the problems associated with managing ontologies in distributed environments such as the Web. They 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 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.

In [KF01] the authors discuss the problem of ontology versioning based on work done in database schema versioning and program interface versioning. They also propose building blocks for the two important aspects of a versioning mechanism: ontology identification and change specification.

[KKOF02] discusses 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 it 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. The paper describes the system and presents the mechanism that we 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.

2.6.3 Evolution and Versioning in Database Systems

According to [SR03], schema evolution has three well-defined and inter-related activities:

1. Core schema evolution, which includes identifying and incorporating changes to the schema while preserving the consistent state of the schema as well as propagating the changes to the data associated with the schema,

2. Version management, which deals with the management of different versions of a schema introduced by schema changes

3. Application management, which examines how applications dependent on the schema may continue to work.

The consistent state of a schema is typically defined using a formal structure such as a graph.

Whereas schema evolution in relational databases is only poorly supported, with the appearance of object-oriented database systems, schema evolution became a research issue. A very early prototype of an object-oriented system that provides support for schema evolution is ORION[BKKK87]. An important part of this work is the development of a formal taxonomy of schema changes and a framework for managing schema changes in object-oriented systems. The semantics of each schema change is examined and a set of invariant properties of the schema is proposed which must be preserved across schema changes.

Object-oriented schema evolution is more relevant for the ontology evolution due to two reasons:

- the object-oriented database models provide a semantically richer model than the relational database system and, therefore, can be considered as the extension of the relation database evolution;

- the object-oriented database models are more similar to the ontology models due to the complex inheritance hierarchies.

In [NK03] the authors compare the evolution of ontologies with the evolution of database schemas. They state that the similarities between database-schema evolution and ontology evolution allow to build on the extensive research in schema evolution. They also identify important differences between database schemas and ontologies. The differences stem from different usage paradigms, the presence of explicit semantics, and different knowledge models. A lot of problems that existed only in theory in database research come to the forefront as practical problems in ontology evolution. These differences have important implications for the development of ontology evolution frameworks: The traditional distinction between versioning and evolution is not applicable to ontologies. There are several dimensions along which compatibility between versions must be considered. The set of change operations for ontologies is different.

[AFM03] presents two perspectives on modelling dynamic information using description logics: In a first part, the authors present a general temporally enhanced conceptual data model able to represent time varying data. In a second part, they introduce an object-oriented conceptual data model enriched with schema change operators, which are able to represent the explicit temporal evolution of the schema while maintaining a consistent view on the instantiated (static) data. Both conceptual data models and their inference problems are encoded in Description Logics.

The problem of the schema evolution and of the schema versioning support has been extensively studied in relational and database papers. [Rod95] provides an excellent survey of the main issues concerned. A semantic approach to the specification and management of object-oriented databases with evolving schemata in introduced in [FGM00]. The authors formalize a generic object-oriented model for the schema versioning and evolution, define the semantics of schema changes and show how interesting reasoning tasks can be supported. This approach is very similar to the declarative approach for the semantics of change in [Sto04b] since both of them can deal with arbitrary complex changes and allow the formal checking of the evolution. A sound and complete axiomatic model for dynamic schema evolution in object-based systems is described in [Pz97]. This is the first effort in developing a formal basis for the schema evolution research. The approach takes into account the key features of types and inheritance. The model can infer all schema relationships from two sets associated with each type.

Mapping Evolution

Mappings are an established paradigm to achieve interoperability in information systems applications. Mappings allow to translate data from one representation to another. As in dynamic environments data sources may change not only their data but also their schemas, their semantics, and their query capabilities. Such changes must be reflected in the mappings. Mappings left inconsistent by a schema change have to be detected and updated. Possible application scenarios for evolving mappings include, but are not limited to, data

integration, model management, or local mappings between peers in Peer-to-Peer information management systems. Related to schema versioning, it is also possible to treat the new version of a schema as a "view" by providing the corresponding mapping.

[VMP03] presents a framework and a tool (ToMAS) for automatically adapting mappings as schemas evolve. Our approach considers not only local changes to a schema, but also changes that may affect and transform many components of a schema. The authors consider a comprehensive class of mappings for relational and XML schemas with choice types and (nested) constraints. Their algorithm detects mappings affected by a structural or constraint change and generates all the rewritings that are consistent with the semantics of the mapped schemas. The approach explicitly models mapping choices made by a user and maintains these choices, whenever possible, as the schemas and mappings evolve. The algorithms have been implemented in a mapping management and adaptation tool ToMAS.

[MP02] presents an approach to schema evolution, which combines the activities of schema integration and schema evolution into one framework. It builds on previous work on a general framework to support schema transformation and integration in heterogeneous database architectures, which relies on the hypergraph-based data model (HDM) as a common data model. In this paper the authors show how this framework also supports evolution of source schemas, allowing the global schema and the query translation pathways to be easily repaired, as opposed to having to be regenerated, after changes to source schemas.

2.6.4 Evolution and Versioning for Other Paradigms

Maintenance of Knowledge-Base Systems

Research in the ontology evolution can also benefit from the many years of research in knowledge-based system evolution [Men99]. There are a vast number of techniques (e.g. knowledge refinement, theory revision, validation and verification, etc.) for assisting the development of knowledge-based systems. They follow the paradigm of detecting problems in a knowledge-based system and suggesting repairs. Most of them attempt to correct errors when a knowledge-based system is being developed.

A common technique for the knowledge maintenance is to reflect over dependencies between knowledge elements. The question is how to represent the dependencies between them. There are three different approaches: (i) procedural approach, (ii) logical approach, and (iii) network approach. All of them are based on the dependency graph analysis. The differences between these approaches are discussed in [MD00].

2.7 Conclusion and Recommendations

We have presented an overview of state of the art in ontology evolution. We have shown a variety of approaches that concern the evolution process, frameworks, methods and tools.

Many of these approaches build on top of or extend the work that has been done in the database and related communities.

However, there are still many open research questions, among them:

Language-independent ontology evolution: The existing ontology evolution approaches heavily depend on the underlying ontology language. One way to address this problem is to model all aspects of the ontology evolution declaratively. This abstraction may assist in the design of a language independent ontology evolution system.

Request specification: A declarative language for the specification of a request for a change would be desirable. It would allow expressing the ontology changes and constraints in a single framework, and, thus, would allow to reason about interactions between the two. This language would differ from the existing ontology query languages, which are only used for the retrieval of the data from an ontology. It would extend these languages by incorporating the modifications, as well.

Ontology dependency: Future research can be directed towards providing more ways for working with multiple ontologies. There are various dependency forms, such as ontology mapping, ontology merging, ontology alignment and ontology integration. For example, the ontology mapping relates similar (according to some metric) concepts and relations from different sources to each other; the ontology merging creates a new ontology from two or more existing ontologies with overlapping parts. Each of these dependency forms puts different requirements on the evolution between dependent ontologies. Some of them can be resolved by introducing a special meta-ontology that captures relationships between entities from different ontologies. For example, to set up the mapping between ontologies, the mapping ontology might be defined. This ontology should contain the equal property that can be used for establishing equivalence between concepts from different ontologies. To support the evolution of the instantiation of this ontology, the ontology evolution approach has to be extended in two ways. Firstly, the set of consistency constraints has to be extended by taking into account the semantics of the mapping ontology. Secondly, the set of changes has to be extended with the more complex changes that can be applied to these mappings. Finally, the evolution support has to take into account that concepts and properties from the ontologies between which the mapping is established are considered as instances in the ontology that describes these mappings.

In the context of this project, the focus of the research will be on developing a framework and methods for the evolution of OWL-based ontologies and their application for data integration scenarios in the presence of heterogeneous evolving data sources.

Chapter 3

Methods for Evolution of OWL Ontologies

The OWL ontology language is a standard for representing ontologies on the Web [HPSvH03], however, the semantics of change operations for OWL has not been considered so far. In this paper, we therefore focus on the evolution of OWL ontologies. More precisely, we consider OWL-DL language, which includes arbitrary sublanguages such as OWL-Lite.

The approach presented in this paper builds partly on our previous work in ontology evolution [Sto04a], which we adapt it towards handling OWL ontologies. The differences are mostly reflected in the ontology consistency definition. As we will show, it does not suffice to define a fixed set of consistency conditions, due to the characteristics of the various sublanguages and the varying usage contexts. Instead, we define the consistency of OWL ontologies at three different levels.

We further define methods for detecting and resolving inconsistencies in an OWL ontology after the application of a change. Finally, as for some changes there may be several different consistent states of the ontology, we define resolution strategies allowing the user to control the evolution.

We exemplarily present resolution strategies for various consistency conditions.

This chapter is organized as follows: We start with a description of the ontology evolution process. We then define the notions of ontology, ontology change operations, and the semantics of change for the ontology model. We further discuss how to detect and resolve structural inconsistency, logical inconsistency and user-defined inconsistency, respectively. The protypical implementation of the methods will be presented in Chapter 4.

3.1 Evolution process

Ontology evolution can be defined as the timely adaptation of an ontology and consistent management of changes. The complexity of ontology evolution increases as ontologies

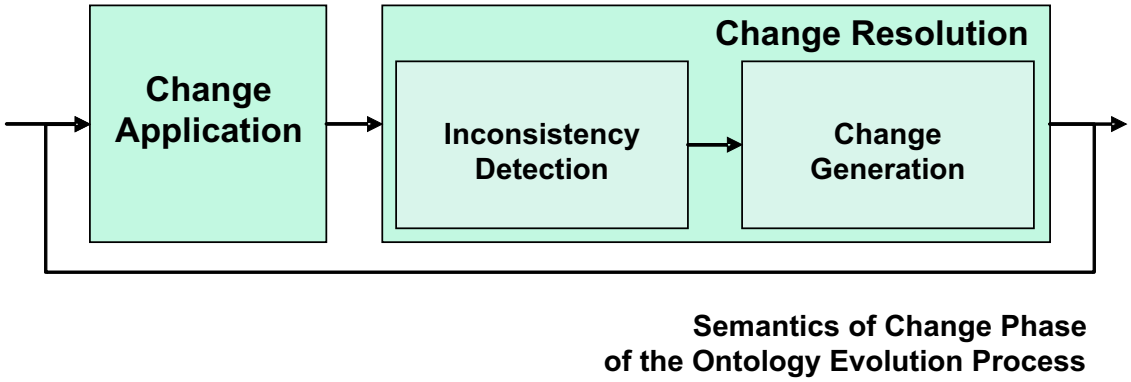

Figure 3.1: A part of the Ontology Evolution Process

grow in size, so a structured ontology evolution process is required. We follow the process described in [Sto04a]. The process starts with *capturing changes* either from explicit requirements or from the result of change discovery methods. Next, in the *change representation* phase changes are represented formally and explicitly. *The semantics of change* phase prevents inconsistencies by computing additional changes that guarantee the transition of the ontology into a consistent state. In the *change propagation* phase all dependent artifacts (ontology instances on the Web, dependent ontologies and application programs using the changed ontology) are updated. During the *change implementation* phase required and induced changes are applied to the ontology in a transactional manner. In the *change validation* phase the user evaluates the results and restarts the cycle if necessary.

In this paper we focus on the semantics of change phase. Its role is to enable the resolution of a given ontology change in a systematic manner by ensuring the consistency of the whole ontology. It is realized through two tasks:

- *Inconsistency Detection*: It is responsible for checking of the consistency of an ontology with the respect to the ontology consistency definition. Its goal is to find ”parts” in the ontology that do not meet consistency conditions;

- *Change Generation*: It is responsible for ensuring the consistency of the ontology by generating additional changes that resolve detected inconsistencies.

The semantics of change phase of the ontology evolution process is shown in Figure 3.1. Changes are applied to an ontology in a consistent state (c.f. Change Application in Figure 3.1), and after all the changes are performed, the ontology must remain consistent (c.f. Semantics of Change in Figure 3.1). This is done by finding inconsistencies in the ontology and completing required changes with additional changes, which guarantee the consistency. Indeed, the updated ontology is not defined directly by applying a requested change. Instead, it is indirectly characterized as an ontology that satisfies a users requirement for a change and it is at the same time a consistent ontology.

In this paper we specifically consider the semantics of change phase for OWL-DL ontologies. Ontology consistency in general is defined as a set of conditions that must

hold for every ontology [Sto04a]. We define the consistency for an OWL-DL ontology at three different levels:

- *Structural Consistency*: Structural consistency makes a determination of the language level of the ontology being examined. It considers only the requirements defined by the ontology language and pays no regard to the internal semantics of an ontology.
- *Logical Consistency*: From the point of view of logic, consistency is an attribute of a (logical) system that is so constituted that none of the facts deducible from the model contradict one other. Therefore, checking the logical consistency of an OWL-DL ontology means determining whether the ontology is consistent in the model-theoretic sense, i.e. whether it is satisfiable.
- *User-defined Consistency*: The user-defined consistency is a set conditions that cannot be derived from the syntax nor semantics of the underlying ontology language. The conditions are explicitly defined by the user and they must be met in order for the ontology to be considered consistent.

We note that most of the existing evolution systems (including the schema evolution systems as well) consider only the structural consistency. The role of an ontology evolution system is not only to find inconsistencies in an ontology and to alert an ontology engineer about them. This is pretty much the kind of support provided by conventional compilers. However, helping ontology engineers notice the inconsistencies only partially addresses the issue. Ideally, an ontology evolution system should be able to support ontology engineers in resolving problems at least by making suggestions how to do that.

Moreover, an inconsistency may be resolved in many ways. In order to help to user to control and customize this process, we have introduced the so-called resolution strategies. Resolution strategies are developed as a method of "finding" a consistent ontology that meets the needs of the ontology engineer. An resolution strategy is the policy for evolution with respect to the his/her requirements. It unambiguously defines the way in which a change will be resolved, i.e. which additional changes will be generated.

In the rest of this paper we formally define different types of consistency and elaborate on how corresponding inconsistency can be detected and resolved.

3.2 Ontology Model and Ontology Change Operations

The goal of ontology evolution is to guarantee the correct semantics of ontology changes, i.e. ensuring that they produce an ontology conforming to a set of consistency conditions. The set of elementary ontology change operations – and thus the consistency conditions – depends heavily on the underlying ontology model. Most existing work on ontology evolution builds on frame-like or object models, centered around classes, properties, etc. However, as in this work we focus on the evolution of OWL-DL ontologies, we follow the axiom-centered ontology model, heavily influenced by Description Logics. In this

section, we will first review the ontology model, define change operations for this model, and describe the semantics of change.

3.2.1 Ontology Model

OWL DL is a syntactic variant of the $\mathcal{SHOIN}(\mathbf{D})$ description logic[HPS04]. The correspondence between the OWL DL abstract syntax and $\mathcal{SHOIN}(\mathbf{D})$ knowledge bases as well as the reduction of OWL entailment to $\mathcal{SHOIN}(\mathbf{D})$ satifiability has been presented in [HPS04]. In the following we will adhere to the more compact, traditional $\mathcal{SHOIN}(\mathbf{D})$ syntax, which we review in the following:

We use a set of concept names N_C, sets of abstract and concrete individuals N_{I_a} and N_{I_c}, respectively, and sets of abstract and concrete role names N_{R_a} and N_{R_c}, respectively. An *abstract role* is an abstract role name or the inverse S^- of an abstract role name S (concrete roles do not have inverses). In the following, we assume that $\mathbf{D}$ is an admissible concrete domain.

An *RBox* $\mathcal{R}$ consists of a finite set of transitivity axioms $\mathsf{Trans}(R)$, and role inclusion axioms of the form $R \sqsubseteq S$ and $T \sqsubseteq U$, where R and S are abstract roles, and T and U are concrete roles. The reflexive-transitive closure of the role inclusion relationship is denoted with $\sqsubseteq^*$. A role not having transitive subroles (w.r.t. $\sqsubseteq^*$, for a full definition see [HST00]) is called a *simple* role.

The set of $\mathcal{SHOIN}(\mathbf{D})$ *concepts* is defined by the following syntactic rules, where A is an atomic concept, R is an abstract role, S is an abstract simple role, $T_{(i)}$ are concrete roles, d is a concrete domain predicate, a_i and c_i are abstract and concrete individuals, respectively, and n is a non-negative integer:

$$\begin{array}{rcl} C & \rightarrow & A \mid \neg C \mid C_1 \sqcap C_2 \mid C_1 \sqcup C_2 \mid \exists R.C \mid \forall R.C \mid \geq n\, S \mid \leq n\, S \mid \{a_1, \ldots, a_n\} \mid \\ & & \mid \geq n\, T \mid \leq n\, T \mid \exists T_1, \ldots, T_n.D \mid \forall T_1, \ldots, T_n.D \\ D & \rightarrow & d \mid \{c_1, \ldots, c_n\} \end{array}$$

A *TBox* $\mathcal{T}$ consists of a finite set of concept inclusion axioms $C \sqsubseteq D$, where C and D are concepts; an *ABox* $\mathcal{A}$ consists of a finite set of concept and role assertions and individual (in)equalities $C(a)$, $R(a, b)$, $a \approx b$, and $a \not\approx b$, respectively.

We denote the set of all possible ontologies with $\mathfrak{O}$. A $\mathcal{SHOIN}(\mathbf{D})$ *knowledge base* $(\mathcal{T}, \mathcal{R}, \mathcal{A})$ consists of a TBox $\mathcal{T}$, an RBox $\mathcal{R}$, and an ABox $\mathcal{A}$. For the direct model-theoretic semantics of $\mathcal{SHOIN}(\mathbf{D})$ we refer the reader to [HST00]. For most of the paper, we do not need to distinguish between $\mathcal{A}$, $\mathcal{R}$ and $\mathcal{T}$ and will therefore call an ontology $\mathcal{O}$ the set of axioms in $\mathcal{T}$, $\mathcal{R}$, and $\mathcal{A}$, i.e. $\mathcal{O} = \mathcal{A} \cup \mathcal{R} \cup \mathcal{T} \in \mathfrak{O}$.

Example 1 As a running example, we will consider a simple ontology modelling a small research domain, consisting of the following axioms:
$Researcher \sqsubseteq Person$, $Student \sqsubseteq Person$ *(students and researchers are persons),*
$Article \sqsubseteq Publication$ *(articles are publications),* $\top \sqsubseteq \forall author.Person$ *the range of author are persons,* $\top \sqsubseteq \forall author^-.Publication$ *(the domain of author are publications),*

$Article(anArticle)$ *(anArticle is an article),* $Researcher(pha)$, $Researcher(lst)$ *(pha and lst are researchers),* $author(anArticle, pha)$, $author(anArticle, lst)$ *(pha and lst are authors of anArticle).*

3.2.2 Ontology Change Operations

Based on the ontology model, we can now define ontology change operations.

Definition 1 *An* ontology change operation *oco* $\in$ *OCO is a function oco* : $\mathfrak{O} \rightarrow \mathfrak{O}$.

For the above defined ontology model of $\mathcal{SHOIN}(\mathbf{D})$, we allow the atomic change operations of adding and removing axioms, which we denote with α^+ and α^-, respectively. Obviously, representing changes at the level of axioms is very fine-grained. However, based on this minimal set of atomic change operations, it is possible to define more complex, higher-level descriptions of ontology changes. Composite ontology change operations can be expressed as a sequence of atomic ontology change operations. The semantics of the sequence is the chaining of the corresponding functions: For some atomic change operations $\text{oco}_1, ..., \text{oco}_n$ we can define $\text{oco}_{composite}(x) = \text{oco}_n \circ ... \circ \text{oco}_1(x) := \text{oco}_n(...(\text{oco}_1))(x)$.

For example, the complex change of removing a concept can be expressed using a composite ontology change operation that removes all axioms which reference this concept.

3.2.3 Semantics of Change

The semantics of change refers to the effect of the ontology change operations and the consistent management of these changes. The consistency of an ontology is defined in terms of consistency conditions, or invariants that must be satisfied by the ontology. We then define rules for maintaining these consistency conditions by generating additional changes.

Definition 2 (Consistency of an Ontology) *We call an ontology* $\mathcal{O}$ consistent *with respect to a set of consistency conditions* $\mathcal{K}$ *iff for all* $\kappa \in \mathcal{K}$, $\mathcal{O}$ *satisfies the consistency condition* $\kappa(\mathcal{O})$.

The consistency conditions may be expressed for example as logical formulas or functions. At this point, we do not make any restriction with respect to the representation of the consistency conditions.

In the following, we will further distinguish between structural, logical and user-defined consistency conditions: $\mathcal{K}_S$, $\mathcal{K}_L$, and $\mathcal{K}_U$, respectively. We will call an ontology *structurally consistent*, *logically consistent* and *user-defined consistent*, if the respective consistency conditions hold for the ontology.

Change Generation If we have discovered that an ontology is inconsistent, i.e. some consistency condition is violated, we need to resolve these inconsistencies by generating additional changes that lead to a consistent state.

Definition 3 *A resolution function $\varrho \in \mathcal{P}$ is a function $\varrho : \mathfrak{O} \times OCO \rightarrow OCO$, which returns for a given ontology and an ontology change operation an additional change operation (which may be composite).*

A trivial resolution function would be a function which for a given ontology and change operation simply returns the inverse operation, which effectively means a rejection of the change. Obviously, for a consistent input ontology, applying a change followed by the inverse change will result in a consistent ontology.

In general, there may be many different ways to resolve a particular inconsistency, i.e. different resolution functions may exist. We can imagine a resolution function that initially generates a set of alternative potential change operations, which may be presented to the user who decides for one of the alternatives. Such a resolution function that depends on some external input is compatible with our definition of a resolution function.

We can now define the notion of an resolution strategy:

Definition 4 (Resolution Strategy) *A* resolution strategy RS *is a total function $RS : \mathcal{K} \rightarrow \mathcal{P}$ that maps each consistency condition to a resolution function. Further we require that for all possible ontologies $\mathcal{O} \in \mathfrak{O}$ and for all oco $\in OCO$ and all $\kappa \in \mathcal{K}$, the assigned resolution function $\varrho = RS(\kappa)$ generates changes oco$' = \varrho(\mathcal{O}, oco)$, which – applied to the ontology oco$'(\mathcal{O})$ result in an ontology that satisfies the consistency condition κ.*

The resolution strategy is applied for each ontology change operation in straightforward manner: As long as there are inconsistencies with respect to a consistency condition, we apply the corresponding resolution function.

Please note that a resolution function may generate changes that violate other consistency conditions (resulting in further changes, that in turn may violate the previous consistency condition). When defining a resolution strategy, one therefore has to make sure that the application of the resolution strategy terminates, either by prohibiting that a resolution function introduces inconsistencies with respect to any defined consistency condition, or by other means, such as cycle detection.

In the following chapters we will introduce various evolution strategies to maintain structural, logical and user-defined consistency of an ontology.

3.3 Structural Consistency

Structural consistency considers constraints that are defined for the ontology model with respect to the constructs that are allowed to form the elements of the ontology (in our case the axioms). However, in the context of OWL ontologies, there exist various sublanguages (sometimes also called species), such as OWL-DL, OWL-Lite, OWL-DLP [Vol04a]. These sublanguages differ with respect to the constructs that are allowed and

can be defined in terms of constraints on the available constructs. The role of these sublanguages is to be able to define ontologies that are "easier to handle", either on a syntactic level to for example allow easier parsing, or on a semantic level to trade some of the expressivity for decreased reasoning complexity. It is thus important that the ontology evolution process provides support for dealing with defined language sublanguage: When an ontology evolves, we need to make sure that an ontology does "not leave its sublanguage".

Because of the variety of the sublanguages, it is not possible to operate with a predefined and fixed set of structural consistency conditions. Instead, we allow allow to define sublanguages in terms of arbitrary structural consistency conditions along with the corresponding reolution functions that ensure that an ontology change operation does not lead out of the defined sublanguage.

3.3.1 Structural Consistency Conditions

We will in the following define what it means for an ontology to be structurally consistent with respect to a certain ontology sublanguage. A sublanguage is defined by a set of constraints on the axioms. Typically, these constraints disallow the use of certain constructs or the way these constructs are used.

Some constraints can be defined on a "per-axiom-basis", i.e. they can be validated for the axioms individually. Other constraints restrict the way that axioms are used in combination.

Consistency Condition for the OWL-Lite sublanguage We now consider OWL Lite as a sublanguage of OWL-DL show how it can be defined in terms of a set of structural consistency conditions $\mathcal{K}_S$[1]:

- $\kappa_{S,1}$ disallows the use of disjunction $C \sqcup D$,
- $\kappa_{S,2}$ disallows the use of negation $\neg C$,
- $\kappa_{S,3}$ restricts the use of the concept $C \sqcap D$ such that C and D be concept names or restrictions,
- $\kappa_{S,4}$ restricts the use of the restriction constructors $\exists R.C$, $\forall R.C$ such that C must be a concept name,
- $\kappa_{S,5}$ limits the values of stated cardinalities to 0 or 1, i.e. $n \in \{0, 1\}$ for all restrictions $\geq n\, R$, $\leq n\, R$,
- $\kappa_{S,6}$ disallows the usage of the oneOf constructor $\{a_1, \ldots, a_n\}$.

[1]Please note that the constraints for the OWL-DL language are already directly incorporated into the ontology model itself.

Other examples for useful structural constraints are for example : (1) Horness, a property which requires that an axiom can be translated into a Horn formula, and thus e.g. restricts disjunction on the right side of the inclusion axiom, (2) disallowing the use of existential qualifiers, which allows us to introduce unnamed objects, or (3) disallow the use of equality.

3.3.2 Resolving Structural Inconsistencies

Once we have discovered inconsistencies with respect to the defined sublanguage, we have to resolve them. An extreme solution would be to simply remove the axioms that violate the constraints of the sublanguage. This would certainly not meet the expected requirements. A more advanced option is to try to express the invalid axiom(s) in a way that is compatible with the defined sublanguage. In some cases, it may be possible to retain the semantics of the original axioms.

Resolution Strategies for OWL Lite In the following we will present a possible resolution strategy for the OWL Lite sublanguage by defining one resolution function for each of the above consistency conditions. Although OWL Lite poses many syntactic constraints on the syntax of OWL DL, it is still possible to express complex descriptions using syntactic workarounds, e.g. introducing new class names and exploiting the implicit negation introduced by disjointness axioms. In fact, using these techniques, OWL Lite can fully capture OWL DL descriptions, except for those containing individual names and cardinality descriptions greater than 1 [HPSvH03].

- $\varrho_{s,1}$ replaces all references to a concept $C \sqcup D$ with references to a new concept name $CorD$, and adds the following axiom: $CorD \equiv \neg(\neg C \sqcap \neg D)$,
- $\varrho_{s,2}$ replaces all references to a concept $\neg C$ in an added axiom with references to a new concept name $NotC$, and adds the following two axioms: $C \equiv \exists R.\top$ and $NotC \equiv \forall R.\bot$, where R is a newly introduced role name,
- $\varrho_{s,3}$ replaces all references to a concept C (or D), where C (or D) is not a concept name, in concepts $C \sqcap D$ with references to a new concept name aC (or aD), and adds the following axiom: $aC \equiv C$ (or $aD \equiv D$),
- $\varrho_{s,4}$ replaces all references to a concept C (where C is not a concept name) in restrictions $\exists R.C$ or $\forall R.C$ with references to a new concept name aC, and adds the following axiom: $aC \equiv C$.

While these first four resolution functions simply apply syntactic tricks while preserving the semantics, there exist semantics-preserving resolution functions for the consistency conditions $\kappa_{S,5}$ and $\kappa_{S,6}$.

However, we can either try to approximate the axioms, or in the worst case, simply remove them to ensure structural consistency. We can thus define:

- $\varrho_{s,5}$ which replaces all cardinality restrictions $\geq n\, R$ with restrictions $\geq 1\, R$ and removes all axioms containing cardinality restrictions $\leq n\, R$,
- $\varrho_{s,6}$ which replaces all references to the concept $\{a_1, \ldots, a_n\}$ with $\top$.

Example 2 Suppose we wanted to add to the ontology from Example 1 the axiom $Publication \sqsubseteq \exists author.\neg Student$, i.e. stating that for all publications must have an author who is not a student. As this axiom violates consistency condition $\kappa_{S,2}$, resolution function $\varrho_{s,2}$ would generate a composite change that adds the following semantically equivalent axioms instead: $Publication \sqsubseteq \exists author.NotStudent$, $Student \equiv \exists R.\top$, $NotStudent \equiv \forall R.\bot$, resulting in a structurally consistent ontology.

3.4 Logical Consistency

While the structural consistency is only concerned about whether the ontology conforms to certain syntactic constraints, the logical consistency addresses the question whether the ontology is "semantically correct".

3.4.1 Definition of Logical Consistency

The semantics of the $\mathcal{SHOIN}(\mathbf{D})$ description logic is defined via a model-theoretic semantics, which explicates the relationship between the language syntax and the model of a domain: model-theoretic semantics. An interpretation $I = (\triangle^I, \cdot^I)$ consists of a domain set $\triangle^I$, disjoint from the datatype domain $\triangle^I_{\mathbf{D}}$, and an interpretation function $\cdot^I$, which maps from individuals, classes and roles to elements of the domain, subsets of the domain and binary relations on the domain, respectively[2]. An interpretation $\mathcal{I}$ satisfies an ontology $\mathcal{O}$, if it satisfies each axiom in $\mathcal{O}$. Axioms thus result in semantic conditions on the interpretations. Consequently, contradicting axioms will allow no possible interpretations.

We can thus define a consistency condition for *logical consistency* κ_L that is satisfied for an ontology $\mathcal{O}$ if $\mathcal{O}$ is satisfiable, i.e. if $\mathcal{O}$ has a model. Please note, that because of the monotonicity of the logic, an ontology can only become inconsistent by adding axioms: If a set of axioms is satisfiable, it will still be satisfiable when any axiom is deleted. Therefore, we only need to check the consistency for ontology change operations that add axioms to the ontology.

Example 3 Suppose, we start out with the ontology from our Example 3.3.2, i.e. the initial example extended with the axiom $Student \sqsubseteq \neg Researcher$ (Students and Researchers are disjoint). This ontology is logically consistent.

Suppose we now wanted to add the axiom $Student(pha)$, stating that the individual pha is a student. Obviously, this ontology change operation would result in an inconsistent ontology, as we have stated that students and researchers are disjoint on the one hand, and that pha is a student and a researcher on the other hand.

[2]For a complete definition of the interpretation, we refer the reader to [HPS04].

Now, there may be many ways how to resolve this inconsistency. One possibility would be to reject the change $Student(pha)$. Alternatively, we could also remove the assertion $Researcher(pha)$. However, if both of these assertions are correct, the user may not be happy with either decision. The most intuitive one may be to retract the axiom $Student \sqsubseteq \neg Researcher$, but also this may not satisfy the user. A further, more complex change, would be to introduce a new class $PhdStudent$, which need not be disjoint with researchers.

3.4.2 Resolving Logical Inconsistencies

In the following, we will present resolution functions that will allow us to define resolution strategies to ensure logical consistency. The goal of these resolution functions is to determine a set of axioms to remove to obtain a logically consistent ontology with "minimal impact" on the existing ontology. Obviously, the definition of minimal impact may be depend on the particular user requirements. A very simple definition could be that the number of axioms to be removed should be minimized. More advanced definitions could include a notion of confidence or relevance of the axioms. Based on this notion of "minimal impact" we can define an algorithm that generates a minimal number of changes that result in a maximal consistent subontology.

However, in many cases it will not be feasible to resolve logical inconsistencies in a fully automated manner. We therefore also present a second, alternative approach for resolving inconsistencies that allows the interaction of the user to determine which changes should be generated. Unlike the first approach, this approach tries to localize the inconsistencies by determining a minimal inconsistent subontology.

Alternative 1: Finding a consistent subontology

In our model we assume that the ontology change operations should lead from one consistent ontology to another consistent ontology. If an ontology change operation (adding an axiom, α^+) would lead to an inconsistent ontology, we need to resolve the inconsistency by finding an appropriate subontology $\mathcal{O}' \subset \mathcal{O}$ (with $\alpha \in \mathcal{O}'$) that is consistent.

Definition 5 (Maximal consistent subontology) *An ontology $\mathcal{O}'$ is a* maximal consistent subontology *of $\mathcal{O}$, if $\mathcal{O}' \subseteq \mathcal{O}$ and $\mathcal{O}'$ is logically consistent and every $\mathcal{O}''$ with $\mathcal{O}' \subset \mathcal{O}'' \subseteq \mathcal{O}$ is logically inconsistent.*

Intuitively, this definition states that no axiom from $\mathcal{O}$ can be added to $\mathcal{O}'$ without losing consistency. In general, there may be many maximal consistent subontologies $\mathcal{O}'$. It is up to the resolution strategy and the user to determine the appropriate subontology to be chosen.

The main idea is that we start out with the inconsistent ontology $\mathcal{O} \cup \{\alpha\}$ and iteratively remove axioms until we obtain a consistent ontology. Here, it is important how we determine which axioms should be removed. This can be realized using a *selection function*. The quality of the selection function is critical for two reasons: First, as we

have to search the power set of axioms in $\mathcal{O}$ for the maximal consistent ontology and thus need to find that *relevant* axioms that cause the inconsistency. Second, we need to make sure that we only remove *dispensable* axioms. (Please note that a more advanced strategy could consider to only remove parts of the axiom.)

The first problem of finding the axioms that cause the inconsistency can be addressed e.g. using a notion of syntactic relevance by analyzing how the axioms are structurally connected:

We can realize a selection functions based on *structural connectedness*:

Definition 6 (Connectedness) *Given a set of axioms $\mathcal{O}$, two axioms α and β are* directly connected *– denoted with connected(α, β) –, if there exists an ontology entity $e \in N_C \cup N_{I_a} \cup N_{I_c} \cup N_{R_a} \cup N_{R_c}$ that occurs in both α and β.*

The second problem of only removing dispensable axioms requires more semantic selection functions. These semantic selection functions can for example exploit information about the confidence in the axioms that allows us to remove less probable axioms. Such information is for example available in probabilistic ontology models, such as [DP04], but will not be considered in this paper.

In the following, we present an algorithm (c.f. Algorithm 1) for finding (at least) one maximal consistent subontology using the definition of structural connectedness (c.f. Definition 6): We maintain a set of possible candidate subontologies Ω, which initially

Algorithm 1 Determine consistent subontology for adding axiom α to ontology $\mathcal{O}$

```
Ω := {O ∪ {α}}
while there exists no O' ∈ Ω such that O' is consistent do
  Ω' := ∅
  for all O' ∈ Ω do
    for all β1 ∈ O' \ {α} do
      if there is a β2 ∈ ({α} ∪ (O \ O')) such that connected(β1, β2) then
        Ω' := Ω' ∪ {O' \ {β1}}
  Ω := Ω'
```

contains only $\mathcal{O} \cup \{\alpha\}$. In every iteration, we generate a new set of candidate ontologies by removing one axiom β_1 from each candidate ontology that is structurally connected with α or an already removed axiom (in $\mathcal{O} \setminus \mathcal{O}'$), until at least one of the candidate ontologies is a consistent subontology. The termination is guaranteed based on the fact that once we have removed all axioms from $O \cup \{\alpha\}$ that are transitively connected with α, the ontology again must be consistent (provided that α itself is consistent and $\mathcal{O}$ was consistent before adding α). As we remove exactly one axiom from each candidate ontology in one iteration, the resulting ontology will not only be maximal with respect to the above definition, but also maximal with respect to cardinality. The corresponding resolution function $\varrho_{L,1}$ thus generates changes that remove the minimal set of axioms to ensure consistency: $\mathcal{O} \setminus \mathcal{O}'$, where $\mathcal{O}'$ is the maximal consistent ontology.

Alternative 2: Localizing the inconsistency

In the second alternative, we do not try to find a consistent ontology, instead we try to find a minimal inconsistent ontology, i.e. a minimal set of contradicting axiom. We call this process *Localizing the inconsistency*. Once we have localized this minimal set, we present it to the user. Typically, this set is considerably smaller than the entire ontology, such that it will be easier for the user to decide how to resolve the inconsistency.

Definition 7 (Minimal inconsistent subontology) *An ontology $\mathcal{O}'$ is a* minimal inconsistent subontology *of $\mathcal{O}$, if $\mathcal{O}' \subseteq \mathcal{O}$ and $\mathcal{O}'$ is inconsistent and for all $\mathcal{O}''$ with $\mathcal{O}'' \subset \mathcal{O}' \subseteq \mathcal{O}$, $\mathcal{O}''$ is consistent.*

Intuitively, this definition states that the removal of any axiom from $\mathcal{O}'$ will result in a consistent ontology.

Again using the definition of connectedness, we can realize an algorithm (c.f. Algorithm 2) that is guaranteed to find a minimal inconsistent ontology: We maintain a set Ω with candidate ontologies, which initially only consist of the added axiom $\{\alpha\}$. As long as we have not found an inconsistent subontology, we add one structurally connected axiom to each candidate ontology.

Algorithm 2 Localize inconsistency introduced by adding axiom α to ontology $\mathcal{O}$

$\Omega := \{\{\alpha\}\}$
while there exists no $\mathcal{O}' \in \Omega$ such that $\mathcal{O}'$ is inconsistent **do**
  $\Omega' := \emptyset$
  for all $\mathcal{O}' \in \Omega$ **do**
    for all $\beta_1 \in \mathcal{O}$ **do**
      if there is a $\beta_2 \in \mathcal{O}'$ such that $connected(\beta_1, \beta_2)$ **then**
        $\Omega' := \Omega' \cup \{\mathcal{O}' \cup \{\beta_1\}\}$
  $\Omega := \Omega'$

Because of the minimality of the obtained inconsistent ontology, it is sufficient to remove one of the axioms to resolve the inconsistency. The minimal inconsistent ontology can be presented to the user, who can select the appropriate axiom to remove. It may be possible that one added axiom introduced multiple inconsistencies. For this case, the above algorithm has to be applied iteratively.

Example 4 We will now show how Algorithm 2 can be used to localize the inconsistency in our running example, which has been introduced by adding the axiom α $Student(pha)$. Applying the algorithm, we start out with the candidate ontologies $\Omega := \{\{Student(pha)\}\}$. Adding the structurally connected axioms, we obtain:
$\Omega := \{\{Student(pha), Researcher(pha)\}, \{Student(pha), Student \sqsubseteq Person\}$,
$\{Student(pha), Student \sqsubseteq \neg Researcher\}, \{Student(pha), Student(lst)\}$,
$\{Student(pha), author(anArticle, pha)\}\}$. *All of these candidate ontologies are still consistent. In the next iteration, adding the structurally connected axiom*

$Student \sqsubseteq \neg Researcher$ to the candidate ontology $\{Student(pha), Researcher(pha)\}$ will result in the minimal inconsistent subontology $\{Student(pha), Researcher(pha), Student \sqsubseteq \neg Researcher\}$.

3.5 User-defined Consistency

The user-defined consistency takes into account particular user requirements that need to be expressed "outside" of the ontology language itself. That means, the semantics of the consistency conditions cannot be described directly in terms of the syntax or semantics of the OWL ontology model: While an ontology may be structurally consistent (e.g. be a syntactically correct ontology according to a particular OWL sublanguage) and may be logically consistent, it may still violate some user requirements.

We can identify two types of the user-defined consistency conditions: (1) *generic conditions* that are applicable across domains and for example represent best design practice or modeling quality criteria, and *domain dependent conditions* that take into account the semantics of a particluar formalism of the domain. One such example are consistency conditions for the OWL-S process model [SASS04] to verify web service descriptions.

In the following we exemplarily show how user-defined consistency condition sand corresponding resolutions function can be described to ensure *modeling quality conditions*. Such modeling quality conditions cover redundancy, misplaced properties, missing properties, etc. We refer the reader to [Sto04a] for a complete reference.

One example of redundancy is *concept hierarchy redundancy*. If a direct superconcept of a concept can be reached through a non-direct path, then the direct link is redundant. We can thus define a consistency condition that disallows concept hierrachy redundancy: $\kappa_{U,1}$ is satisfied if for all axioms $C_1 \sqsubseteq C_n$ in $\mathcal{O}$ there exist no axioms in $\mathcal{O}$ with $C_1 \sqsubseteq C_2$, ..., $C_{n-1} \sqsubseteq C_n$.

We can further define a corresponding resolution function $\varrho_{U,1}$ that ensures this consistency condition, which generates a change operation that removes the redundant axiom $C_1 \sqsubseteq C_2$.

Example 5 Suppose, we start out with the ontology from our Example 3.3.2, i.e. the initial example extended with the axiom $Professor \sqsubseteq Person$ (a professor is a person). This ontology is consistent with respect to the consistency definition $\kappa_{U,1}$.

Suppose we now want to add the axiom $Professor \sqsubseteq Researcher$, stating that the a professor is a researcher. Obviously, this ontology change operation would result in an ontology that is inconsistent with respect to $\kappa_{U,1}$ since there is an alternative path (through the concept $Researcher$) between the concept $Professor$ and its direct superconcept $Person$. The resolution function $\varrho_{U,1}$ would generate a change operation that removes the axiom $Professor \sqsubseteq Person$.

3.6 Conclusion

In this chapter we have presented an approach to formalize the semantics of change for the OWL onology language (for OWL-DL and sublanguages in particular), embedded in a generic process for ontology evolution. Our formalization of the semantics of change allows to define arbitrary consistency condititions – grouped in structural, logical, and user-defined consistency – and to define resolution strategies that assign resolution functions to that ensure these consistency conditions are satisfied as the ontology evolves. We have shown examplarily, how such resolution strategies can be realize for various purposes.

Chapter 4

Prototypes

In this chapter we will present three prototypes for ontology management and evolution:

- The KAON tool suite comprises tools for the engineering, management, and evolution of ontologies. Although not primarily developed in the SEKT project, it does have a special role in the project and is therefore included in this chapter: 1) It is used by the project partners for the engineering and management of their ontologies, and 2) it serves as the basis for further developments of ontology management and evolution prototypes in the project.
- *dlponvert* is a tool that aims at closing the gap between the Description Logics based ontology languages and logic programming languages. It allows the conversion of a subset of OWL to Datalog and thus serves as the basis for efficient reasoning with OWL ontologies.
- *evOWLution* is a software component that builds on KAON2[1] to support the consistent evolution of OWL ontologies using the methods described in chapter 3.

4.1 KAON

KAON consists of a number of different modules providing a broad bandwidth of functionalities centered around **creation, storage, retrieval, maintenance and application of ontologies**. It was and currently is being further developed in a joint effort mainly by members of the Institute AIFB at University of Karlsruhe and the FZI – Research Center for Information Technologies, Karlsruhe.

Before presenting an outline of this document we will clarify in the next section the overall picture on what kind of KAON components exist currently. If you are not yet confused by the plethora of tools or if you have only an interest in a special tool you can leave out the next section.

[1] `http://kaon2.semanticweb.org/`

In this chapter we will only provide an overview of the KAON tool suite. In Appendix A we will present detailed information about download, and usage of KAON.

Note: Please be aware that we here present a snapshot of currently available versions. Future versions of the tools might have additional and/or different functionalities etc. In appendix A.4 we will provide detailed information about download and installation including a table describing the version numbers of the here described tools.

The *KArlsruhe ONtology and Semantic Web tool suite* a.k.a. **KAON ToolSuite** is, as mentioned before, an Open Source ontology management infrastructure. However, there exists also external components which support functionalities such as e.g. ontology learning from texts. An overview of the KAON ToolSuite and its main components - *KAON* and *KAON Extensions*.

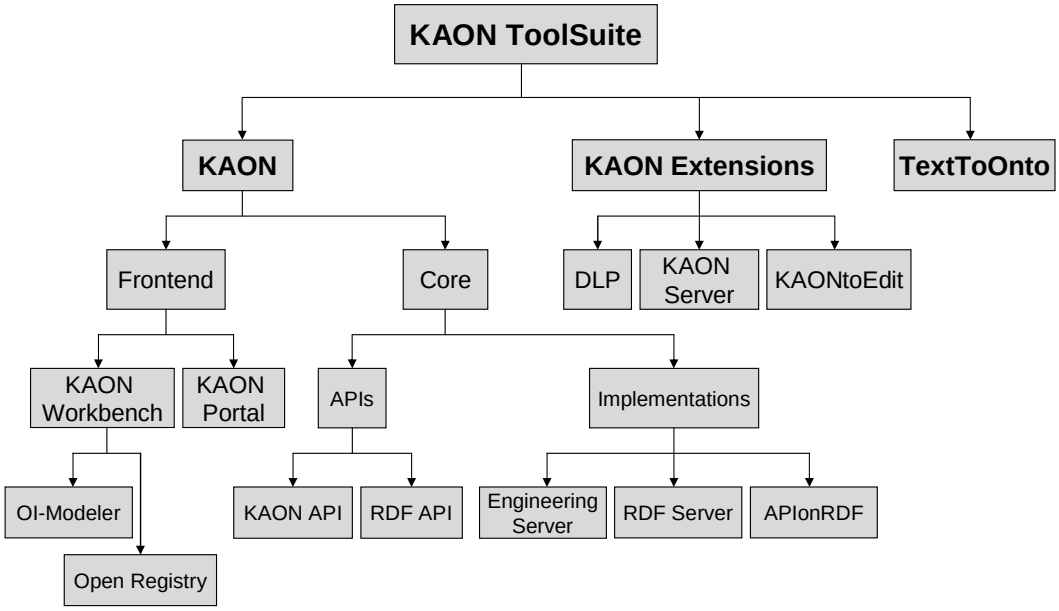

Figure 4.1: KAON Tool Overview

- **KAON** consisting of *KAON Frontend* and *KAON Core* includes a variety of different modules for ontology creation and management.

  The **Frontend** is represented by two applications developed in order to be used particularly by human users:

  - **KAON Workbench** provides a graphical environment for ontology-based applications. It includes the **OI-Modeler** (cf. chapter A.1) - a graphical ontology editor - and the **Open Registry** (a.k.a. *Ontology Registry*), which provides mechanisms for registering and searching ontologies in a distributed context.
  - **KAON Portal** is a simple tool for multi-lingual, ontology-based Web portals.

The **Core** of KAON supports programmatic access to ontologies by including both *APIs* and *implementations* for managing local and remote ontology repositories:

  - An abstract interface for accessing various types of ontologies independently of the regarding storage mechanisms is provided by the **KAON API** and the **RDF API** (cf. chapter A.2).
  - Currently three different **implementations** of the KAON API and the RDF API are available: Whereas the **Engineering Server** (cf. chapter A.3) is an ontology server using a scalable database representation of ontologies, the **RDF Server** can be used for storing and accessing RDF models. **APIonRDF** (cf. chapter A.2) is a main-memory implementation of the KAON API on the RDF API.

- **The KAON Extensions** are a collection of optional components not included in the standard distribution of KAON.

  - **DLP** (*Description Logic Programs*) supports efficient ontology reasoning by mapping Description Logic into Logic Programs.
  - **KAON Server** can be considered as Application Server for the Semantic Web, which provides a generic infrastructure to facilitate plug'n'play engineering of ontology-based applications.
  - **KAONtoEdit** is a plug-in for OntoEdit [oG03], which allows to work directly on implementations of the KAON API in order to load, modify and store KAON ontology models.

- **TextToOnto** is a KAON-based tool suite supporting the ontology engineering process by providing a collection of independent tools for ontology learning and maintenance.

KAON Architecture While we have so far provided an overview on the components and tools that are part of or related to KAON, we now want to focus on the rather technical interplay of some of those components, i.e. we intend to give a coarse outline of KAON's functional architecture distinguishing between APIs, implementations of those APIs and data sources to be accessed.

Figure 4.2 illustrates some of the interactions between KAON's APIs and reference implementations and highlights the central role of the KAON API. Each client, e.g. KAON's ontology editor OI-Modeler, accesses KAON ontologies and instances independently of the storage mechanism via KAON API. In so doing, the OI-Modeler for example employs KAON's ontology evolution facilities integrated into KAON API.

As already mentioned, APIonRDF represents an in-memory implementation of the KAON API to access RDF-based data sources via the RDF API. For the RDF API in turn

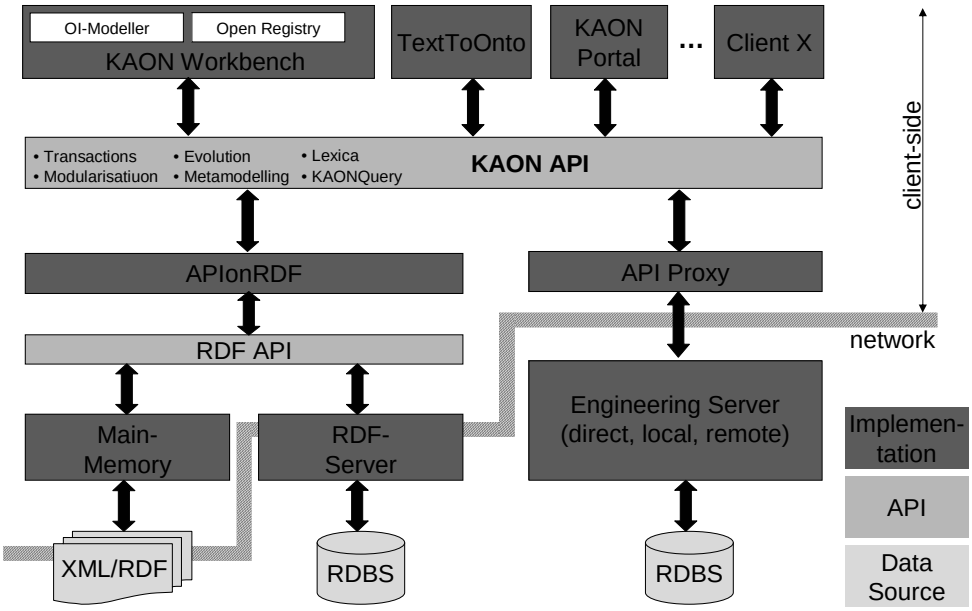

Figure 4.2: KAON Architecture Overview

exist two reference implementations: On the one hand, a simple main memory implementation including RDF parser and serializer. On the other hand, the RDF Server which implements the RDF API remotely and allows for persistently storing RDF ontology models in relational databases and hence enables transactional ontology modification.

Sketched on the right-hand side, the API Proxy depicts an implementation of the KAON API that acts as a client-side proxy for various types of the KAON Engineering Server (cf. Chapter A.3). That Engineering Server, being accessed remotely via an API Proxy, features mechanisms to store KAON ontologies in relational databases, to distribute change notifications (thus allowing for multi-user ontology engineering), and to bulk-load ontology elements.

For a more thorough and detailed depiction of KAON's architecture the interested reader is referred to [Vol04b, MMV02].

4.2 dlpconvert

4.2.1 Motivation for DLP

In the past few years, the W3C concentrated (and still works) on defining the main building blocks of the Semantic Web. With RDF [MM04] they provided a common data model and afterwards they defined different ontology definition languages, RDFS [BG04] (offering only basic semantics) and the Web Ontology Language OWL [SWM04].

But all of these standards are still very new, and thus only very limited support of tools for inferencing and querying exists. But this capabilities will proof crucial for the

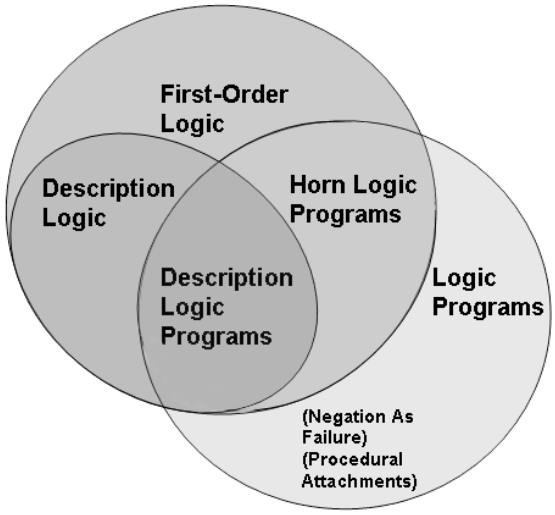

Figure 4.3: Expressive Overlap of DL with LP [Vol04b]

development of semantically enabled technologies.

Regarding RDFS or the OWL fragment OWL Full, both are undecidable. But also OWL DL and even OWL Lite yield a very high computational complexity, and thus efficient reasoners able to deal with these fragments in a sufficient scalable way may take some time to come along.

But in SEKT we want to work with real life systems, and we want to do it now. Embracing the idea of "small can be beautiful" [Rou04], we can, with a little limitation, which in practise will only very rarely limit the actual modelling possibilities, gain access to a strong and almost immediate tool support.

In order to tame the computational complexity, it was already decided within the consortium to remain in OWL Lite if possible. But by choosing DLP[2], we gain even better computational characteristics - and more important, we gain access to various already implemented tools. The DLP fragment and its computational implications are described in much more detail in [Vol04b].

DLP is the intersection of the sets of semantic expressible axioms in description logic and logic programming. It imposes further constraints on OWL DL, in order to guarantee that all axioms stated in DLP will be automatically transformable in an efficient way to logic programming rules and facts. This logic programs may then be evaluated efficiently with systems like XSB or Ontobroker. These systems are already implemented and can be used out of the box.

As [Vol04b] analysed, 99% of the axioms in the ontologies taken from the daml.org repository of ontologies were within the DLP fragment. Especially simple taxonomical descriptions usually made when using OWL Lite anyway, are within DLP. Light weight ontologies like the SEKT upper level ontology PROTON [TKM04] will hardly ever leave the DLP fragment. Only some complex class constructors like arbitrary cardinality con-

[2] `http://logic.aifb.uni-karlsruhe.de`

straints are not expressible, but most of them are not part of OWL DL anyway.

A second major advantage of DLP, besides its computational features, is its future stability. Currently there exist two major trends for ontology representation, description logic – with OWL being the most visible one – and logic programming, particularly F-Logic. Both have benefits and drawbacks and allow for different usage scenarios.

Even though the W3C can generate big impact, due to its role in the standardisation of web technologies, we can't know how OWL will develop in future years.

DLP on the other hand provides maximal flexibility, as it can easily be translated from one paradigm to the other. We can even decide to use the best fitting approach based on the task at hand. It remains fully reusable, as DLP is a proper subset of the W3C standard OWL.

4.2.2 `dlpconvert`

In order to facilitate the the use of DLP, and to show its full potential right away, `dlpconvert` is provided. It may be found on `http://logic.aifb.uni-karlsruhe.de/dlpconvert/`, and allows to convert OWL DL encoded DLP fragments into logic programming syntaxes.

`dlpconvert` is based on the algorithms for reducing description logics to datalog implemented in KAON2[3] and described in [HMS04b] and, in greater length, in [HMS04a]. It reads an OWL ontology, reduces it to disjunctive Datalog and finally serialises it into a logical program, which can be used for easier reading and thus understanding by people with an appropriate logic background or as input for logic interpreters like XSB.

`dlpconvert` is provided in two ways: as a command line tool with numerous switches, providing different ways of name transformations and serialisation options. It can be used to convert an OWL DL file directly into a Prolog program file, that can be consumed by a Prolog interpreter as it is.

Besides the command line tool, on the website there is an online conversion available. You may choose to either supply an URL for an ontology, upload a file from your local hard disk or even write (or copy and paste) an ontology directly onto the website. Your ontology will be converted and the result shown to you as a HTML page.

4.2.3 Example

Let's take an example to see the advantages of `dlpconvert`. In philosophy, the probably most classical example for inferencing is the following syllogism:

All humans are mortal.

Socrates is a human.

Therefore Socrates is mortal.

This syllogism, along with another fact surrounding Socrates' - that the author of the *Politeia* is actually Plato, and not Socrates, is formalized in the following OWL file:

[3] `http://kaon2.semanticweb.org`

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE Ontology [
 <!ENTITY ex
 "http://logic.uni-karlsruhe.de/dlpconvert/example1#">
]>

<owlx:Ontology owlx:name="&ex;"
    xmlns:owlx="http://www.w3.org/2003/05/owl-xml#">

<owlx:Class owlx:name="#human" owlx:complete="false">
    <owlx:Class owlx:name="#mortal"/>
</owlx:Class>

<owlx:Individual owlx:name="#Socrates">
  <owlx:type owlx:name="#human" />
</owlx:Individual>

<owlx:ObjectProperty owlx:name="#isauthor">
    <owlx:domain owlx:class="#human"/>
    <owlx:range owlx:class="#book"/>
</owlx:ObjectProperty>

<owlx:Individual owlx:name="#Plato">
  <owlx:ObjectPropertyValue owlx:property="#isauthor">
    <owlx:Individual owlx:name="#Politeia" />
  </owlx:ObjectPropertyValue>
</owlx:Individual>

</owlx:Ontology>
```

If you go to the website `http://logic.aifb.uni-karlsruhe.de/dlpconvert/`, the example ontology, that is referenced in the URL line, actually is this one described here. So you may just go to the website, and click on the first submit button, in order to see the following result.

```
mortal(X) :- human(X).
book(Y) :- isauthor(X, Y).
human(X) :- isauthor(X, Y_0).
isauthor(plato, politeia).
human(socrates).
```

Both representations actually have the same meaning. The second is undoubtedly shorter, and for many readers the syntax is much more reader-friendly, and for every

person with a background in logic programming the second representation is immediately understandable.

We can directly feed this result to a prolog engine, like XSB prolog[4]. This way, the XSB system understands the semantics of the original OWL file and we can ask questions about the given ontology.

```
?- mortal(socrates).
yes
```

The system dutifully gives us the correct answer: Socrates is a mortal. And if we ask XSB about the author of the *Politeia*, it will answer correctly. Better yet, it knows that Plato too is a human, because the domain of the *isauthor* relationship is *human* (we never stated explicitly, that Plato is a human).

```
?- isauthor(X, politeia).
X = plato
yes

?- human(plato).
yes
```

This can be used in several way, for example for questioning knowledge bases (as we did) or for checking the consistency of an ontology.

4.2.4 Future Work

`dlpconvert` is just a prototype right now. Quite a number of features are waiting to be implemented:

- better error support. Right now it is not implemented, the user almost does not get any information on what went wrong, if something went wrong
- the website version does not offer any switches yet, but they should be available from the web, too
- an FLogic serialization will provide the conversion of OWL DL encoded DLP knowledge bases to FLogic syntax. This way, tools like OntoBroker will be able to deal with OWL DL effectively
- names. The heuristics to generate names should be more flexible than it is right now. This way we can provide better names, especially from automatically generated ontologies that usually have insignificant names for concepts and instances. Also, namespaces should be considered in order to avoid clashes.
- tests. `dlpconvert` has not yet been properly tested.

[4] `http://xsb.sourceforge.net`

4.3 evOWLution – Evolution of OWL Ontologies

evOWLution is a software component that supports the methods for the consistent evolution of OWL ontologies using the methods described in Chapter 3. It builds on top of the KAON2 ontology management infrastructure which is currently being developed as part of the EU IST project DIP[5].

The implementation includes evolution strategies for various fragments of ontology languages, including OWL-DL, OWL-Lite and OWL-DLP, as well evolution strategies for logical consistency. Additionally it allows to plug-in further evolution strategies for structural consistency (to support additional sublanguages), logical consistency, and user-defined consistency.

The prototype can be downloaded from `http://www.aifb.uni-karlsruhe.de/WBS/pha/owlevolution`. It contains the source code, binaries, and the examples from Chapter 3.

4.3.1 Usage Example

The software is not intended to be a standalone tool, it rather is a component to be integrated in an ontology-based application to support the consistent evolution of the of ontologies using the particular resolution strategies required in the application.

The prototype includes an example of such an application and can be used as a reference for development.

We will now show some small code fragments of this sample application (cf. class `evolution/Evolution.java`) to explain the usage: A resolution strategy is created by instantiating the class `ResolutionStrategy`, we can then register consistency conditions and the corresponding resolution functions:

```
ResolutionStrategy resolutionStrategy = new ResolutionStrategy();
resolutionStrategy.registerElementaryResolutionsStrategy(new
      LogicalConsistencyChecker(),
      new LogicalResolutionStrategy());
resolutionStrategy.registerElementaryResolutionsStrategy(
      new OWLLiteConsistencyChecker(),
      new OWLLiteResolutionStrategy());
```

Suppose we now load an ontology

```
Ontology ontology = connection.openOntology(
      "http://www.aifb.uni-karlsruhe.de/WBS/pha/owlevolution/sample",
       new HashMap<String, Object>());
```

consisting of the following axioms:

[5] see `http://dip.semanticweb.org/`

```
[classMember Researcher pha]
[classMember Researcher lst]
[subClassOf Student Person]
[subClassOf Article Publication]
[subClassOf Researcher Person]
```

We then try to add the axiom:

```
[subClassOf Student [not Researcher]]
```

This will violate the consistency condition defined in `OWLLiteConsistencyChecker`, therefore the resolution function in `OWLLiteResolutionStrategy` will be called to resolve the inconsistency. This is done by transforming the axiom containing the negation into semantically equivalent statements without explicit negation.

We then try to add the axiom

```
[classMember Student pha]
```

which will result in an ontology that violates the consistency condition of `LogicalConsistencyChecker`, therefore the resolution function in `LogicalResolutionStrategy` will be called to resolve the inconsistency. This is done by identifying the minimal inconsistent subontology, which is then presented to the user:

```
[classMember Student pha]
[classMember Researcher pha]
[equivalent notResearcher [all R_Researcher owl:Nothing]]
[equivalent Researcher [some R_Researcher owl:Thing]]
[subClassOf Student notResearcher]
```

The user can decide to remove any of the axioms, resulting in a consistent ontology.

4.3.2 Future Work

We plan to collaborate closely with our project partner Vrije Universiteit Amsterdam to come up with a joint framework which combines ontology evolution and inconsistency reasoning. We believe that a generic framework targets well the problems we want to solve. The future development of our prototypes depends on the results of our collaboration, i.e. on how the framework actually will look like.

Chapter 5

Conclusion

Ontologies play a central role in the research and development of the SEKT project. In the dynamic business environments that we address in this project, ontologies quite naturally change over time. We thus need methods and tools to cope with the the evolution of ontologies.

In this document we have presented a survey on existing work on ontology evolution. This existing work for example includes the evolution infrastructure in the KAON ontology management system based on the KAON ontology model (presented in detail in the appendix of this document). However, in the scope of the SEKT project, we want to move towards using richer, standards-based ontology languages. Therefore, OWL is the language of choice. As OWL is a fairly new ontology language, the semantics of change for OWL ontology has so far been considered only to a limited extent.

Our developmemt of methods and tools therefore has focused on the management and evolution of OWL ontologies. In this document, we have presented a first approach for the consistent evolution of OWL ontologies. We have also presented a first prototype implementing these methods. In the future we will extend these presented methods and tools and integrate them in the KAON2 ontology management infrastructure.

We have further presented a tool that plays an important role for the efficient reasoning with OWL ontologies: `dlpconvert` is a tool that enables the conversion of a large subset of OWL (the DLP fragment) to Datalog, which allows to rely on well-established, efficient logic programming engines for the reasoning with ontologies.

Appendix A

Ontology Management and Evolution in KAON

A.1 Ontology Editor OI-Modeler

OI-Modeler is KAON's tool for ontology creation and ontology maintenance[1]. The OI-Modeler's main goal is to allow scalability for editing large ontologies and to incorporate some basic usability issues related to ontology management and evolution.

The goal of this chapter is to introduce the reader to the OI-Modeler's main features. For that purpose we use a running example about the construction of a tiny ontology, presenting OI-Modeler's basic functionalities. For further details on how to work with the OI-Modeler we refer to "OI-Modeler User's Guide" [Kar02].

This chapter is mainly for end-users of the OI-Modeler. Programmers may want to proceed to the following Chapter A.2 which describes the KAON API and how to access it.

Create New OI-Model

After having launched the KAON Workbench (by invoking `kaongui.bat`, cf. Section A.4) the user may work with the OI-Modeler, KAON's ontology editor. To work with an OI-Model, you can create a new ontology or open an existing one. When choosing "Create new OI-Model" from the "File" menu, a dialog box appears asking for the specifics of the ontology instance to be created (see Figure A.1).

When speaking about working with an OI-Model, it is important to emphasize that the ontology may be stored in different ways depending on the intended application or usage scenario. If a *really* voluminous ontology with lots of concepts and instances shall be created, the usage of the Engineering Server is advisable, since it employs a relational database system to store all entities involved (see Chapter A.3). In general, however, it is

[1]The "OI" here refers to "Ontology Instance". Hence, in the following we refer by the term "OI-Model" to an instance of an ontology.

fully sufficient to store the OI-Model in main memory, in particular when intending to get started with the OI-Modeler. Here, tab "RDF Models" has to be chosen from the dialog shown in Figure A.1 and so the resulting ontology will be stored locally on the user's hard disc drive.

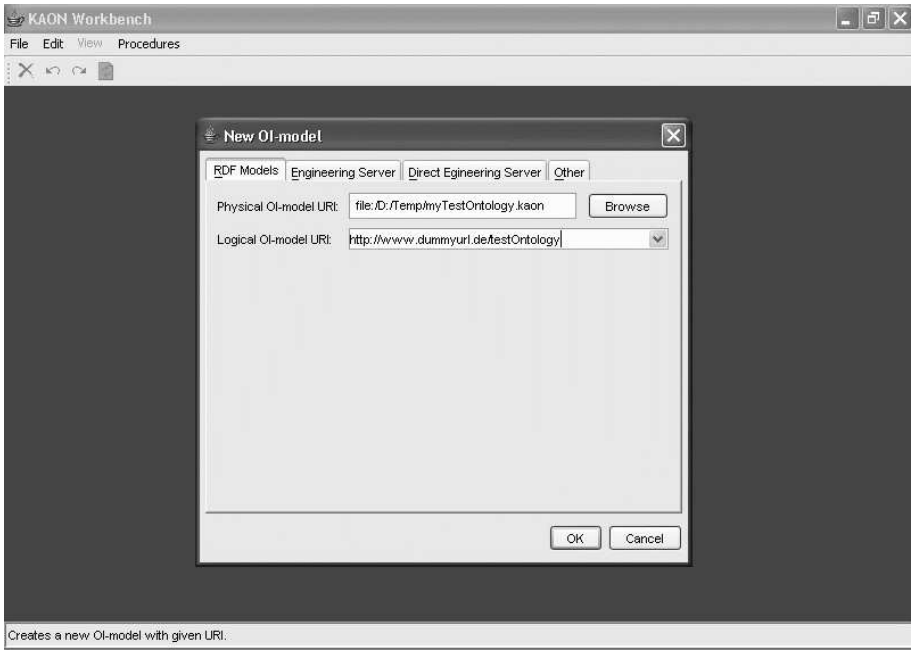

Figure A.1: Creation of a new OI-Model

For our running example, however, we do not want to employ the Engineering Server, instead our ontology shall be stored in main memory. For that reason, we only have to provide a physical as well as a logical[2] URI for our ontology.

As illustrated in Figure A.2, the OI-Modeler provides different views on the Ontology and allows to inspect its components (concepts, instances, properties and lexicon).

Graph The graph in the upper section of the window shows the ontology entities and the connections between them. The graph layout algorithms in OI-Modeler are based on an open-source TouchGraph[3] library.

Each graph node features up to six little arrows (see Figure A.3). By clicking on those arrows related entities can be expanded, so that the user can successively

[2]Each OI-Model has two URIs that uniquely identify it. The *physical* URI is the URI used to access the model. For example, if the model is located in file `D:\Temp\myRunningExampleOntology.kaon`, then the physical URI of the model will be `file:/D:/Temp/myRunningExampleOntology.kaon`. Each OI-Model also has a logical URI, which is independent from the physical one. E.g., a model may have the logical URI `http://kaon.semanticweb.org/myModel.kaon`, although it is not loaded from the web. A model's logical URI should be globally unique, whereas its physical URI typically is not globally unique, and is often relative to the system which processes the model.

[3]`http://www.touchgraph.com`

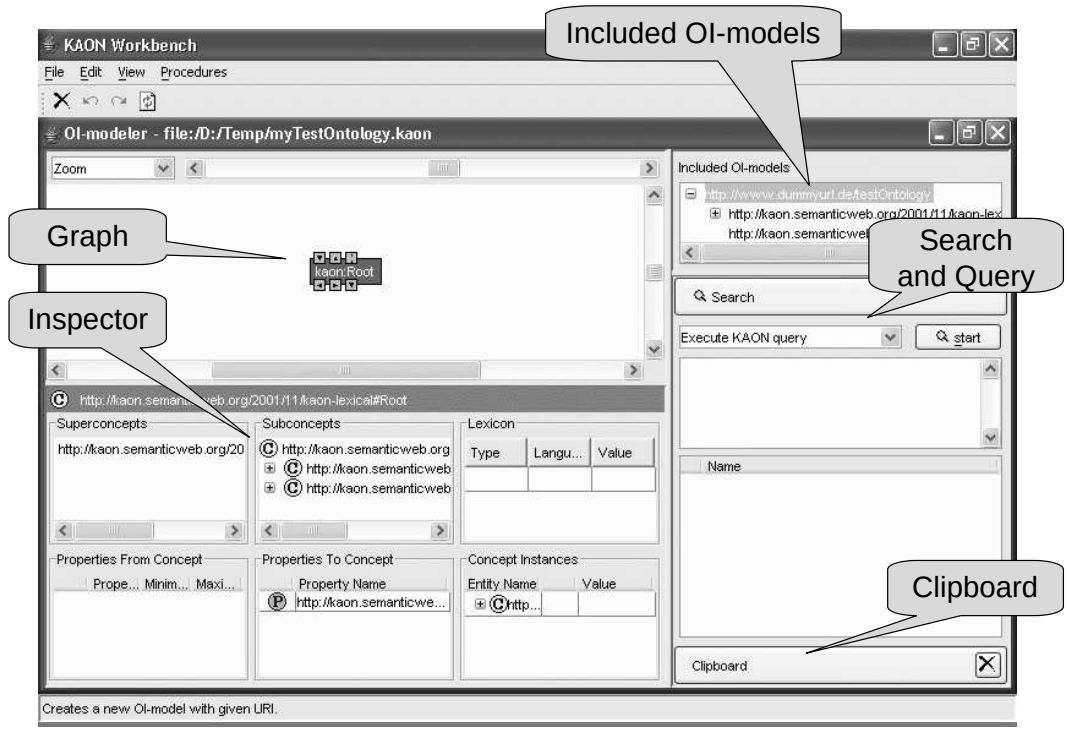

Figure A.2: Main Window of the OI-Modeler

browse through the ontology. For example, for a concept the user may expand that concept's sub- and super-concepts, properties to and from this concept, the concept's instances as well as its spanning instances. Regarding the notion of spanning instances please refer to [MMV02].

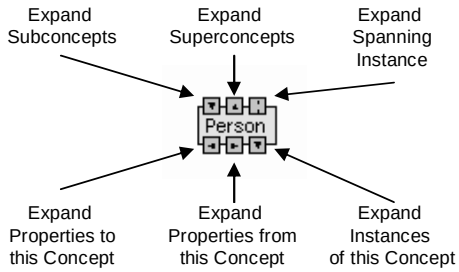

Figure A.3: Characteristics of Nodes in OI-Modeler's Graph Visualization

Inspector In the inspector you can find all information about the ontology entity that is currently selected in the graph. Thus, the inspector's appearance adapts to the type of entity (concept, property, or instance) currently selected. If, for example, a concept is selected information about that concepts and its super- and subconcepts are displayed, also the properties to and from the concept and the concept instances.

Furthermore, the inspector may be used to directly create new (sub-)concepts (see below).

Included OI-Models The OI-Modeler allows for including ontologies. This means that the user is able to combine two (or more) ontologies to one ontology. An OI-Model always consists of two basic or system ontologies: The `kaon-root` and the `kaon-lexical`. To include an OI-Model one can choose "Open and include OI-Model" in the "Edit" menu. Then, a new window opens and the source of the OI-Model to be included can be selected. Please refer to Section A.1.

Search and Query With the search function, one can easily find different named nodes. It is possible to search for concepts, instances, and properties and to perform a keyword-based search for any matching item.

Furthermore, KAON provides a query language KAON Query suited for posing queries to the ontology.

The search and query facilities integrated into OI-Modeler are described in more detail in Section A.1.

Clipboard The Clipboard is for copy and paste use. It allows to copy entities to the clipboard, store them there, and later use them by pasting into the ontology. Please refer to Section A.1 for further details.

Add New Concept

OI-Modeler provides three ways to create a concept. You can add new concepts by

1. using the "Edit" menu and choosing "New Concept...",
2. opening the context-menu (right mouse-button) in the graph window and choosing "New Concept...",
3. using the Inspector and opening the context-menu there.

Figure A.4 illustrates the first of the three above-mentioned ways.

Sub-concepts are thematic refinements of concepts. When intending to add a sub-concept c_s to an existing concept c, first concept c has to be selected – as a consequence, details regarding that concept are displayed in the Inspector. Now, the sub-concept can be added to c with one of the three possible ways mentioned before. In Figure A.5 the third alternative (using the context menu in the Inspector) is shown.

Add New Property

The procedure to add a property to an ontology model is almost the same as creating a concept, i.e. the user can choose between three ways just as in the case of adding

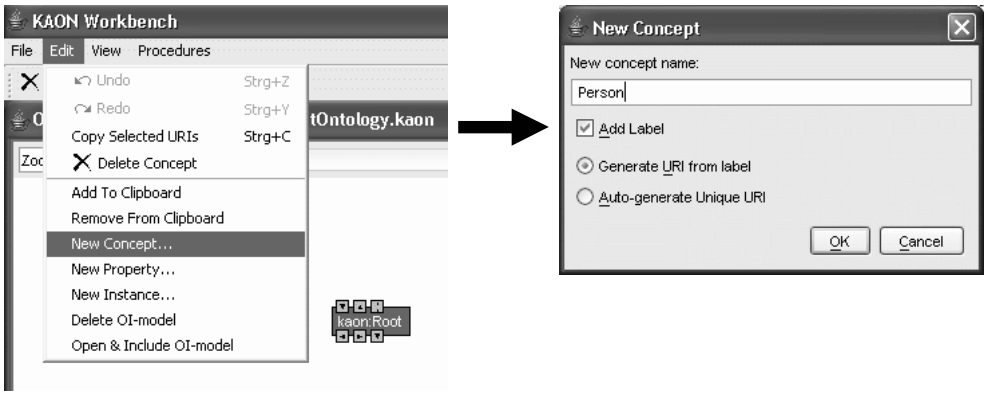

Figure A.4: Adding a Single Concept Person

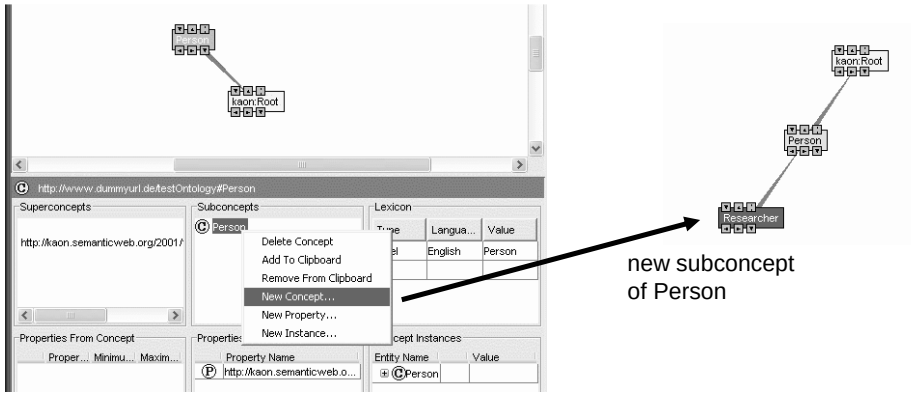

Figure A.5: Adding Sub-Concept Researcher to Concept Person

concepts as described in Section A.1. However, the OI-Modeler differs between two kinds of properties: properties from and to a concept.

Furthermore, when speaking about OI-Modeler's facilities to edit *properties* we ought to clarify what we mean with that term and with terms often used additionally or synonymously such as attribute and relation.

Relation/Relationship is used as a generic term to refer to any kind of property that interlinks concepts.

Properties *from* a Concept are relations to other concepts (instances). In the graph view they are displayed in the same way as *attributes* of a concept.

Properties *to* a Concept are relations from other concepts to the concept under consideration.

Attributes do not connect two or more concepts, but they are rather used to express a certain characteristic of a concept, e.g. a description, long name, or URL. Attributes are often typed as XML Schema data types.

We will use mainly the terms property and attribute. Please note that both are inherited equally from super-concepts to sub-concepts.

In Figure A.6 a later development stage of the ontology is shown. There, you can see what is meant by those different types of properties, how they relate to concepts, and how they are displayed in OI-Modeler's graph view. In the meantime additional concepts (e.g. Paper and Researcher) have been added. Moreover, there are also two new properties: First, there is the HASWRITTEN property which, on the one hand, represents a property from concept Researcher and, on the other hand, a property to concept Paper.

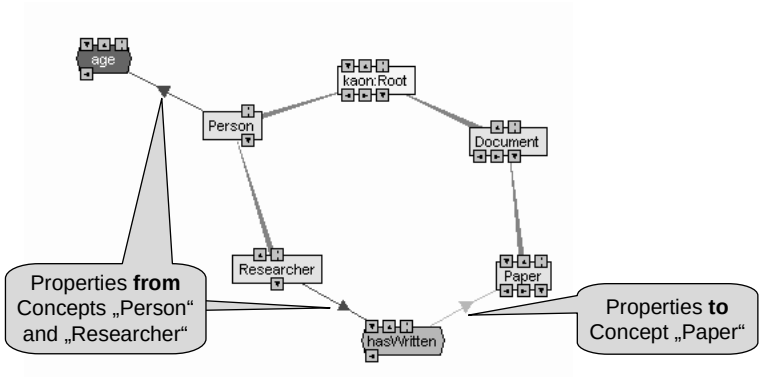

Figure A.6: Properties from and to a Concept

Then, there is the AGE property from concept Researcher. This property represents an attribute of concept Researcher. Note, that when selecting a specific property in the graph view, the Inspector displays the characteristics of that property. Among those, there are also checkboxes that may be used to declare that property to be

- an attribute
- a symmetric property
- a transitive property

So, for the AGE property, the attribute flag has been set.

Moreover, it is possible to specify an inverse property to another one. For example, the inverse property for HASWRITTEN may be called HASAUTHOR being a property *from* concept Paper *to* concept Researcher.

Just as sub-concepts are thematic refinements of concepts, sub-properties represent thematic refinements of properties. OI-Modeler also support the refining of properties by creating sub-properties, whose creation we do not describe in more detail here.

Add New Instance

Just as in the case of concepts and properties, users can add a new instance by three different ways. In each case, however, you first have to choose the concept you want to add the instance to. Then, you may

1. use the “Edit” menu’s entry “New Instance...”
2. use a concept’s context menu (right mouse button) and choosing “New Instance...”
3. use the Inspector’s table “Concept Instances” in the right by making a right-click on the respective concept name to which an instance shall be added and choosing “New Instance...” from the context menu opened.

In any way a new window appears asking the user to provide a name for the instance: Figure A.4 illustrates the third way to create an instance and shows the mentioned window asking for the name. The resulting ontology, after having added two further instance, is sketched in the right part of that figure.

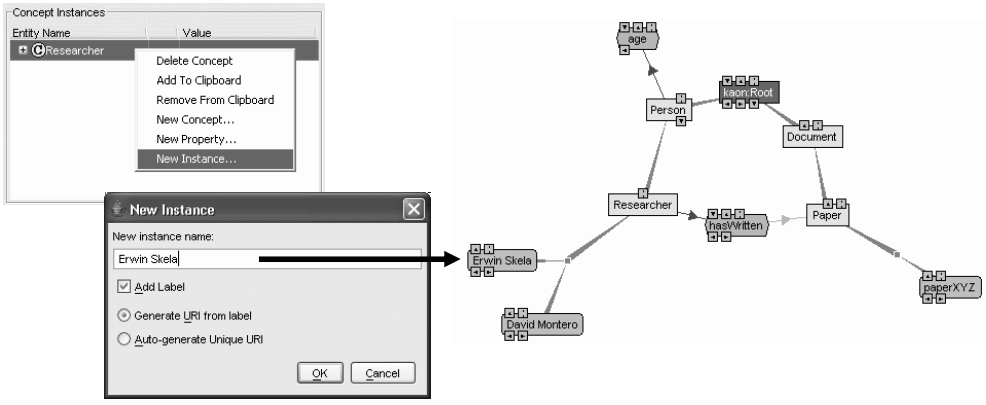

Figure A.7: Creation of a New Instance

Instantiate Properties

The properties between the instances are the relations between these instances.

First, we want to add an attribute instance AGE to the instance `Erwin Skela`. To do so, there are the three usual ways (via “Edit” menu, via context menu for that instance’s graph node, and via context menu for that instance in the inspector). In any way, the menu item “Add Attribute Instance...” has to be chosen. OI-Modeler then present a sub-menu containing all attributes that are defined for the instance’s corresponding concept. In our case (cf. Figure A.8) there is of course only the attribute AGE listed. After having performed that step, an attribute instance is created which initially does not contain a concrete value yet.

To assign a specific value (bottom-right part of Figure A.8) you may edit the input field in the Inspector corresponding to that newly created attribute instance.

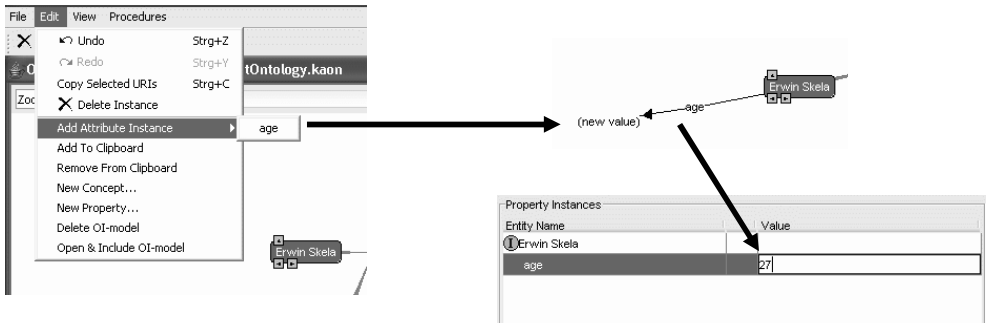

Figure A.8: Instantiating an Attribute

Next, we want to connect to instances using the property HASWRITTEN. To be exact, we want to link the instance `David Montero` with instance `paperXYZ` via the HASWRITTEN property.

There are two different ways to connect instances through a property:

1. Use the graph view and select the (source) instance you want to connect. Then do a right-click on the instance that ought to be connected (target instance). From the opening menu choose "Connect Instances Using", and from its sub-menu the respective property (here: HASWRITTEN).

2. It is also possible to press and hold the left mouse button (on the source instance) and then to drag the cursor to the (target) instance you want to connect. A line appears, as shown in the picture and if you disengage the mouse button, a menu appears and you can choose the property (here: HASWRITTEN).

Both options are visualized in Figure A.9.

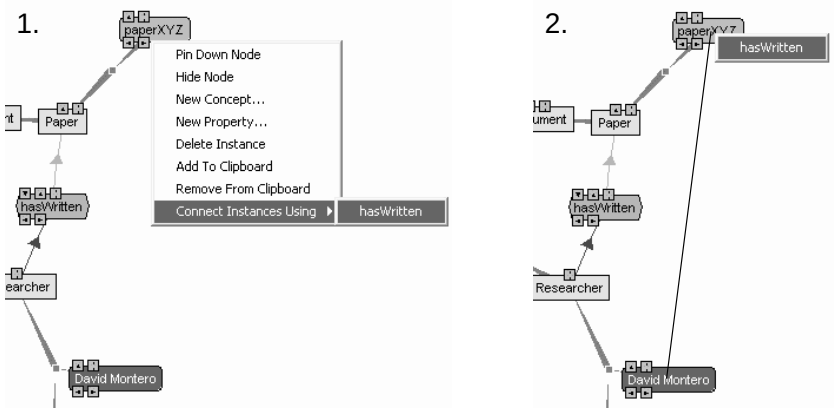

Figure A.9: Instantiating a Property

Delete Concept

To remove a concept you can use one of the three following options. First, select the concept to be deleted, then

1. do a right-click on it and choose “Delete Concept” from the context menu opening,
2. select the concept’s name in the Inspector, do a right-click on it and choose “Delete Concept” from the context menu opening.

Deleting of properties and instances works similar to the here described deletion of concepts.

While adding new elements to an ontology in general does not induce needed follow-on operations, the removal of an entity from the ontology may easily do so. For example, when deleting a certain concept, one upcoming question is how to handle the concept’s instances. As questions like that are of high importance for ontology evolution, the following section is devoted to KAON’s current features concerning ontology evolution.

Evolution Features

Industrial and academic environments are very dynamic, inducing changes to application requirements. Using an ontology-based system, often the underlying ontology must be evolved in order to adapt to those changes. As ontologies grow in size, the complexity of change management increases, thus requiring a well-structured ontology evolution process.

In KAON the user is provided with capabilities to customize and control the process of ontology evolution. It employs so-called evolution strategies that encapsulate certain policies for evolution with respect to the user’s demands (see [SMMS02b] and [SSH02]). As those evolution features are an integrated element of KAON, their usage fully available from within the OI-Modeler.

Note, that evolution reversibility services are provided as special service of KAON API, allowing different applications to reuse these powerful features.

Using Evolution Features

Potentially, an ontology change might corrupt the instances, dependent ontologies as well as application programs running against the ontology and/or the data base. With option “Set-up Evolution Parameters” from the “Procedures” menu the user is allowed to define the strategy how the OI-Modeler handles changes in the ontology, e.g. the deletion of concepts.

The window shown in Figure A.10 shows the parameters by which users of the OI-Modeler may decide for a specific ontology evolution strategy. So, for example, problems are addressed like the handling of orphaned concepts that come into existence after a (parent) concept has been deleted, or the handling of properties that do not have a domain concept any more. The evolution strategies shown are rather self-explanatory.

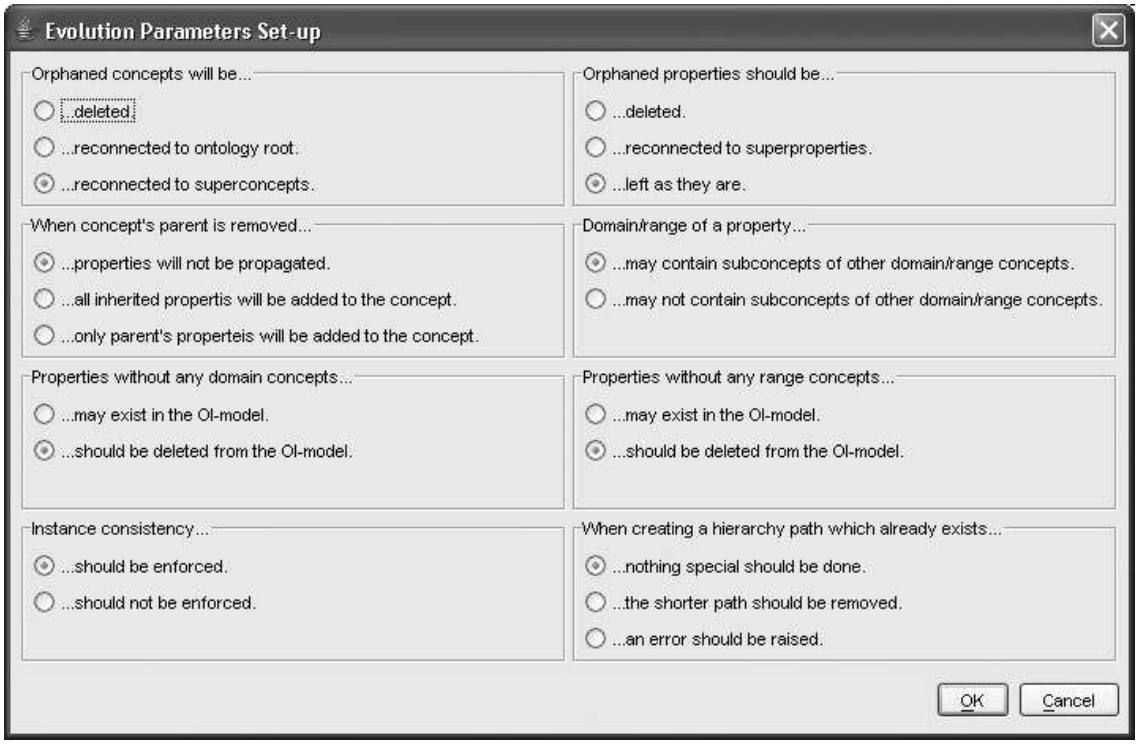

Figure A.10: Setting Up Parameters for Ontology Evolution Strategies

Changes to the ontology are performed by assembling elementary and composite changes into a sequence which is based upon the respective evolution strategy. To ensure atomicity of updates to the ontology (and thus to allow for do/undo functionality), either no or all changes from that sequence have to be processed.

In the "Procedures" menu you find the option "Show Evolution Details". If that option is not checked, changes (e.g. concept deletion) are performed immediately. By checking that option the mentioned extended sequence of changes is presented to the user for approval.

From our current version of the ontology (compare Figures A.7 and A.9) we now intend to delete concept Paper. Then, the dialog shown in Figure A.11 appears and displays the sequence of all (atomic) changes that have to be performed in accordance to the evolution strategy chosen. Obviously, the removal of concept Paper induces the deletion of property HASWRITTEN and hence of its property instantiation (`David Montero` HASWRITTEN `paperXYZ`), the deletion of instance `paperXYZ` and of course the desired deletion of concept Paper.

To further aid the understanding why certain changes have to be performed, related elementary change actions are grouped together in a tree-like structure increasing the understandability of why some changes/side effects have to be executed. After those changes have been reviewed and approved by the user, they are passed to the ontology and performed.

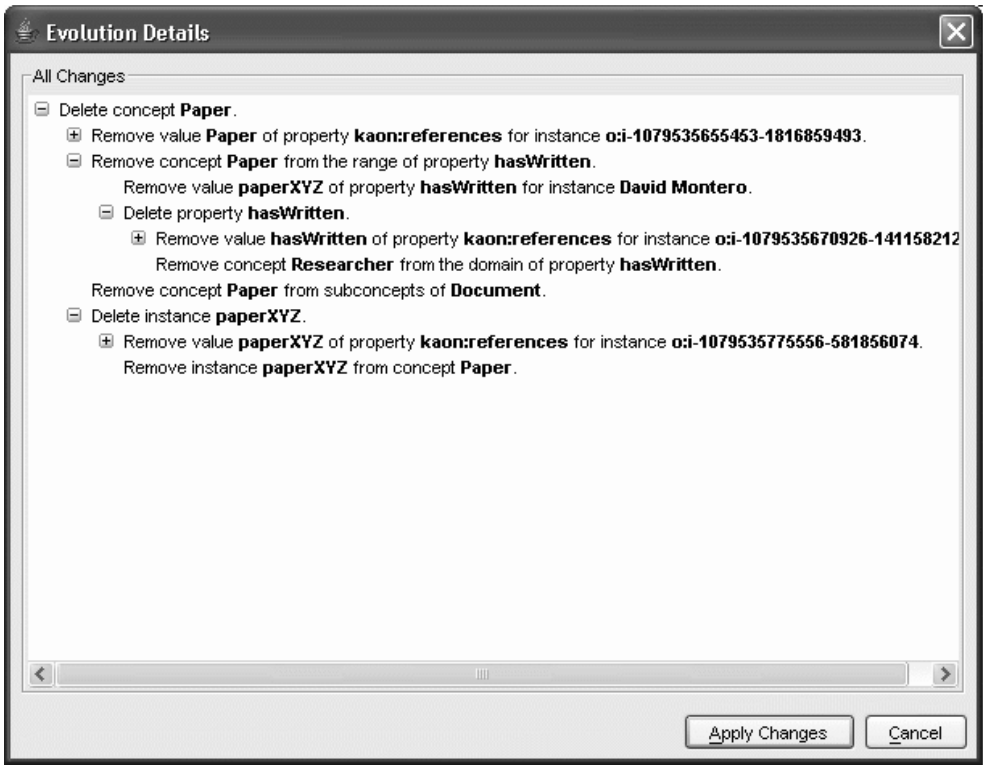

Figure A.11: Presenting Evolution Details to the User for Approval

Undo / Redo Functionality

There are various circumstances under which it may be desirable to reverse the effects of ontology evolution, e.g.

- The ontology engineer may fail to understand the actual effects of his/her changes and may approve a change that actually should not have been performed.
- Sometimes it is helpful to change the ontology for experimental purposes.
- When working collaboratively on an ontology, several ontology engineers may have different ideas on how the ontology ought to evolve.

It is obvious that for each elementary change there is exactly one inverse change that, when applied, reverses the effect of the original change. Based on the infrastructure described in the previous section, it is not hard to realize the requirement for reversibility of ontology engineering actions and to provide an appropriate undo/redo functionality: To reverse the effect of some extended sequence of changes, a new sequence of inverse changes in reverse order needs to be created and applied.

In other words, reversibility means undoing *all* effects of some change, which is in general not the same as requesting an inverse change manually. For example, if a concept

is deleted from a concept hierarchy, its subconcepts will need to be deleted as well, attached to the root concept, or attached to a parent of the deleted concept. Reversing such a change is of course not equal to recreating the deleted concept – one needs, also to revert the concept hierarchy into its original state.

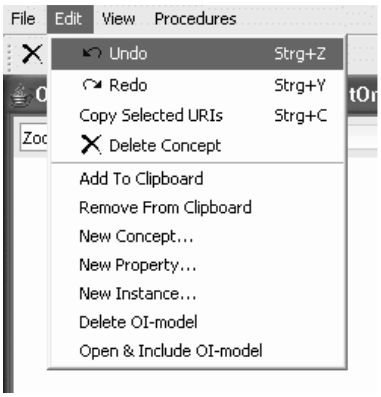

Figure A.12: Undoing and Redoing Changes

In OI-Modeler the undo and redo features are provided via the "Edit" menu as shown in Figure A.12.

Ontology Evolution Log File The problem of reversibility is typically solved by creating evolution logs. An evolution log stores information about each change in the system, allowing to reconstruct the sequence of changes. With each change applied to the ontology the evolution log additionally associates further information [MSSV02], like meta-information such as change description, cost of change, time required to perform the change, cause of the change, or identity of the change's author.

The following excerpt from a log file illustrates some of the information put into that tracking facility. It refers to adding of concept Manual as sub-concept of Document to the ontology.

```
<a:AddEntity rdf:ID="i-1079545047999-1104860243"
    a:inOIModel="http://www.dummyurl.de/testOntology"
    a:version="85"  >
    <a:has_previousChange rdf:resource="#i-1079545047999-24213731"/>
    <a:has_previousHistoryChange rdf:resource="#i-1079545047999-24213731"/>
    <a:has_referenceInstance>
      http://www.dummyurl.de/testOntology#i-1079545046317-1018711561
    </a:has_referenceInstance>
</a:AddEntity>
<a:AddEntity rdf:ID="i-1079545047999-1343051864"
    a:firstChangeInAGroup="true"
    a:has_referenceInstance="http://www.dummyurl.de/testOntology#Manual"
    a:inOIModel="http://www.dummyurl.de/testOntology"
    a:version="85">
    <a:has_previousChange rdf:resource="#i-1079545041129-1524299176"/>
    <a:has_previousHistoryChange rdf:resource="#i-1079545041129-1524299176"/>
</a:AddEntity>
[...]
```

```
<a:AddInstanceOf rdf:ID="i-1079545047999-24213731"
    a:has_referenceConcept="http://www.dummyurl.de/testOntology#Document"
    a:has_referenceInstance="http://www.dummyurl.de/testOntology#Manual"
    a:inOIModel="http://www.dummyurl.de/testOntology"
    a:version="85">
    <a:has_previousChange rdf:resource="#i-1079545047999-1343051864"/>
    <a:has_previousHistoryChange rdf:resource="#i-1079545047999-1343051864"/>
</a:AddInstanceOf>
<a:AddPropertyInstance rdf:ID="i-1079545047999-345568492"
    a:has_referenceProperty=
      "http://kaon.semanticweb.org/2001/11/kaon-lexical#references"
    a:has_referenceTargetInstance="http://www.dummyurl.de/testOntology#Manual"
    a:inOIModel="http://www.dummyurl.de/testOntology"
    a:version="85">
    <a:has_previousChange rdf:resource="#i-1079545047999-2048209500"/>
    <a:has_previousHistoryChange rdf:resource="#i-1079545047999-2048209500"/>
    <a:has_referenceSourceInstance>
      http://www.dummyurl.de/testOntology#i-1079545046317-1018711561
    </a:has_referenceSourceInstance>
</a:AddPropertyInstance>
<a:AddPropertyInstance rdf:ID="i-1079545047999-359720445"
    a:has_referenceProperty="http://kaon.semanticweb.org/2001/11/kaon-lexical#value"
    a:has_referenceTargetObject="Manual"
    a:inOIModel="http://www.dummyurl.de/testOntology"
    a:version="85">
    <a:has_previousChange rdf:resource="#i-1079545047999-345568492"/>
    <a:has_previousHistoryChange rdf:resource="#i-1079545047999-345568492"/>
    <a:has_referenceSourceInstance>
      http://www.dummyurl.de/testOntology#i-1079545046317-1018711561
    </a:has_referenceSourceInstance>
</a:AddPropertyInstance>
```

Inclusion of other OI-Models

As mentioned before OI-Modeler is capable of including managing several ontologies in parallel an of including entire ontologies into another one. The semantics of the inclusion are described in detail in [MMS+03b].

As depicted in Figure A.2 OI-Modeler's main window. Each OI-Model consists per default of two so-called system ontologies, `kaon-lexical` and `kaon-root`. Note, that it is possible to mask these system ontologies (in the graph view) by deselecting the option "System Objects" from the "View" menu.

To include an OI-Model choose "Open and Include OI-Model" in the "Edit" menu. A new window opens and you can select the source of the OI-Model you want to include (see Figure A.13).

Querying and Searching

Queries in KAON (and thus in the OI-Modeler) are an experimental feature that from the perspective of the KAON development team is far from being finished. The primary role of the current support for querying in KAON is to gather feedback in order to improve these features in next versions of KAON. It is quite likely that the query syntax and/or the API will change significantly in the future.

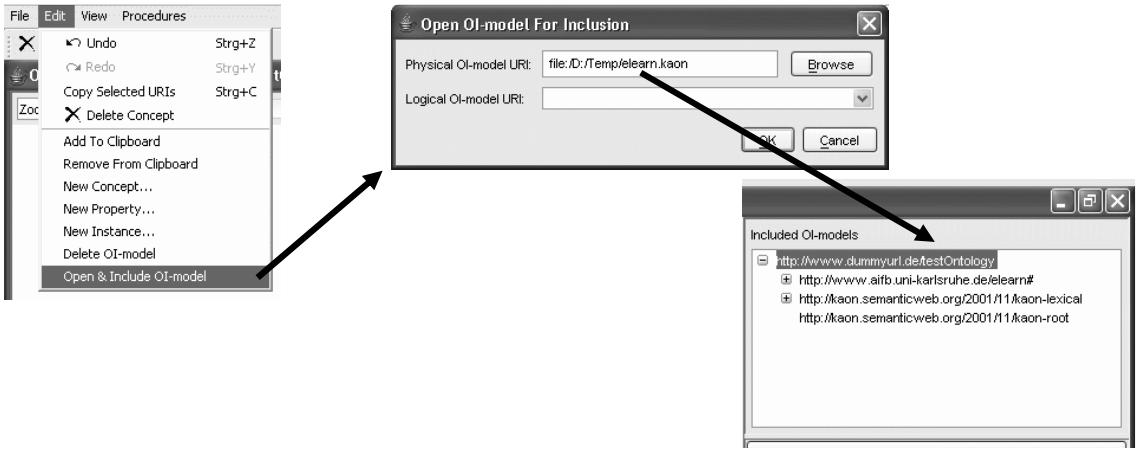

Figure A.13: Inclusion of other OI-Models

KAON provides an experimental conceptual query language (KAON Query) that allows easy and efficient locating of elements in KAON OI-Models. However, as already mentioned queries in KAON are under development, so the interested reader is referred to [KK04].

Keyword-Based Searching With the search function, the user can easily find different named nodes. It is possible to search for

- anything: Every matching entity in the ontology will be displayed.
- concepts: Matching concepts will be displayed.
- instances: Matching instances will be displayed.
- properties: Matching properties will be displayed.

Figure A.14 shows how to search for the keyword Person in our running example – that search returns two results: The concept Person as well as a spanning instance `Person`. For the notion of a spanning instance please refer to [MMV02].

The results are presented as a list matching the search string. In particular, the user can also paste selected results into the “Graph window” via drag & drop.

Using the Clipboard

The clipboard is a convenient way to employ copy & paste functionality when working with an ontology. By opening an entity’s context menu via right-clicking on it (e.g. in the graph view or in the Inspector) and choosing “Add to Clipboard”, or by choosing “Add to Clipboard” from the “Edit” menu, the respective entity is copied to the clipboard (see Figure A.15).

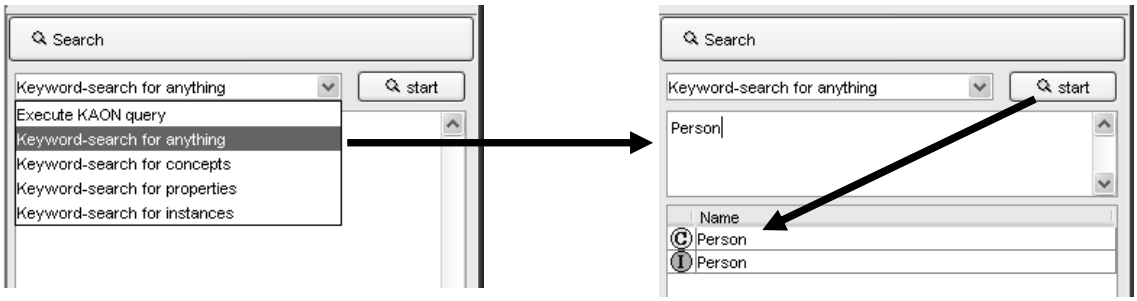

Figure A.14: OI-Modeler's Searching Facility

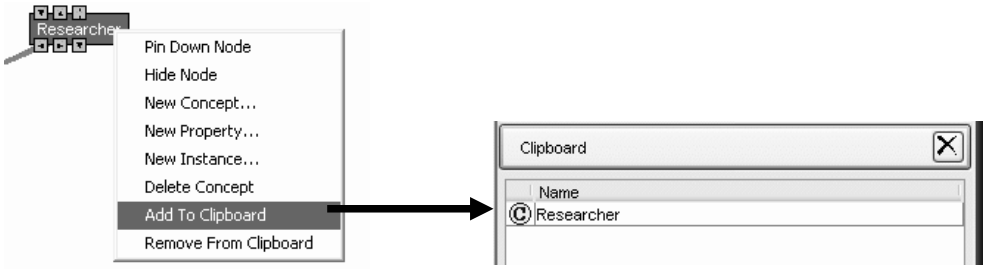

Figure A.15: OI-Modeler's Clipboard

After having been added to the clipboard, the respective entity may be used via drag & drop and can that way be integrated into the graph view or into the Inspector.

For example, the copied concept Researcher may be dragged onto another concept Scientific Staff and would thus be made a sub-concept of that concept.

Other Features

In this section we give a very brief overview of some of OI-Modeler's other features.

Loading/Saving the Workspace The entries "Load Workspace" and "Save Workspace" from the "File" menu allow the user to load/save a workspace containing all windows of the previous/current work session.

Open/Save OI-Model In analogy to creating a new OI-Model (cf. Section A.1) it is possible to load a previously saved OI-Model by choosing the corresponding entry from the "File" menu.

Duplicate OI-Model This option from the "File" saves the current OI-Model under another name and hence duplicated it.

Copy to new OI-Model With this option from the "File" menu you can copy an existing OI-Model into a new one. This replica has the reference to the original OI-Model.

If you choose this function, the same window as in the function "Open OI-Model" appears, and you can choose where to save the backup.

View Specifics The view onto the user interface can be customized (via the "View" menu) to, e.g.

- refresh the graph representation (shortcut F5),
- show only selected nodes (shortcut F4),
- use an "Incremental Search Graph": With that function you can search the graph for e.g. a keyword incrementally,
- show/hide the entire graph view, Inspector, and clipboard,
- hide system objects: An OI-Model consists of three objects: `kaon-lexical#Root`, `kaon-lexical#language`, and `kaon-lexical#LexicalEntry`. By switching of the "System Objects" you only visualize the `kaon-root` and so the ontology gets more clear because less concepts and instances are shown in the graph and the inspector.

Language Parameters Language parameters are to be found in the "View" menu. The user can choose between English, German, French, Spanish, Arabic, and Chinese.

Context Menus As mentioned most entities in OI-Modeler feature context menus whose appearance varies from entity to entity. For a detailed description of context menus we refer to [Kar02].

Entity Icons To easily distinguish concepts, properties, and instances in the Inspector OI-Modeler utilizes several specific icons as shown in Figure A.16.

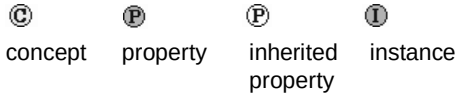

Figure A.16: OI-Modeler's Icons for Entities

Lexical Layer All ontological entities are considered as language neutral in KAON. On the lexical layer, lexical descriptions referring to different entities in the KAON representation vocabulary may be defined. The lexicon is always accessible within the Inspector. A lexical entry is a lexicalization of a concept, attribute, relation, and instance. Several types of lexical entries are defined. The standard lexical description are multilingual labels that may be used for the user's interface. A label is a specific kind of a lexical entry, describing a primary descriptor of an ontological or knowledge base entity. Another kind of lexical entries are morphologically reduced word

stems that may be used by a natural language processing system. shows lexical elements available in OI-Model.

A synonym is a specific kind of a lexical entry, describing synonymous words for an ontological or knowledge base entity. The documentation allows you to enter a text description of the ontological entity. The lexical layer also allows you to create

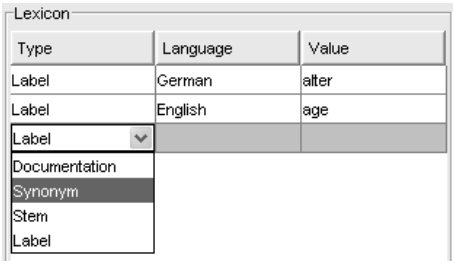

Figure A.17: OI-Modeler's Lexicon

multilingual ontologies. As shown in the picture it is possible to label a concept (in this case the property AGE) in different languages.

A.2 KAON API Description

Overview

The KAON API is a set of interfaces developed in order to offer programmatic access to KAON ontologies by providing classes such as `Concept`, `Property` and `Instance`. Because the API does not make any assumptions about the underlying ontology persistence mechanisms it totally decouples the user from all details of ontology access and storage. It is the concrete implementation of the API, which determines, for example, whether an ontology created with the KAON API will be stored in an RDF file or a local or remote database (cf. subsection A.2). Currently three different implementations of the KAON API are available:

Engineering Server The engineering server (cf. section A.3) is an ontology server using a scalable database representation for storing KAON ontologies. It is optimized for ontology engineering by offering scalable, transactional and concurrent access to ontology information.

RDF Server The RDF server uses the RDF API for storing and accessing RDF models. Although quite similar to the engineering server supporting transactions and multi-user operations, it does not provide any functionalities for conflict detection or bulk-loading.

APIonRDF Implementation The main memory implementation of the KAON API on the RDF API provides in-memory model manipulation for KAON. When you download the standard KAON distribution (cf. Appendix A.4) this is the default setting.

Since the main memory implementation can be considered as the standard implementation of the KAON API this chapter will focus on APIonRDF.

Important Features

Meta Modeling Meta Modeling means that a concept or a property may be considered as an instance of a meta-concept. Such an instance is then called the *spanning-instance* of the regarding concept or property. It can be retrieved by using the `getSpanningInstance()` method, which is defined in the `Entity` interface.

Evolution Strategies Since each change to an ontology might leave the model in an inconsistent state, the KAON API supports the use of evolution strategies (cf. subsection A.2) for computing sequences of additional changes, which are necessary for safely performing the requested change.

Change Notifications Implementing the Observable design pattern the OI-Model interface allows listeners to receive notifications about model updates.

Lexical Layer Lexical information such as labels or documentation can be added to an OI-Model by assigning `LexicalEntry` objects to the instance interpretation of concepts, properties or instances.

Modularization KAON as well as the KAON API support building ontologies modularly by means of ontology inclusion. Each OI-Model may include other OI-Models provided that those are of the same type (e.g. RDF-based or server-based). Because the inclusion is implemented as a link, not as a copy, all changes to the included OI-Model will immediately affect the including OI-Model.

Examples

This section gives a brief introduction on using the KAON API on the basis of a small sample ontology which is also used in chapter A.1.

Select Implementation

The KAONConnection interface is provided by the KAON API in order to separate clients from different API implementations. Each of these implementations must include at least one implementation of KAONConnection, which can be used by clients in order to access the regarding OI-Model implementation.

As long as a user is working with only one implementation of the KAON API instances of KAONConnection may be created directly by using the appropriate constructor (e.g. `new KAONConnectionImpl()`). If an application should work with any or with more than one implementation the KAONManager class has to be used to obtain a KAONConnection object. The following listing A.1 demonstrates how to get a KAONConnection object for APIonRDF.

```
HashMap parameters = new HashMap();
parameters.put( KAONManager.KAON_CONNECTION,
    "edu.unika.aifb.kaon.apionrdf.KAONConnectionImpl" );
KAONConnection connection =
    KAONManager.getKAONConnection( parameters );
```

Listing A.1: KAONConnection (APIonRDF)

Each set of parameters passed to `KAONManager.getKAONConnection` must contain a `KAON_CONNECTION` parameter which determines the type of connection, which is returned. A direct connection to an Engineering Server, for instance, would require the following `KAON_CONNECTION` parameter:

```
parameters.put( KAONManager.KAON_CONNECTION,
"edu.unika.aifb.kaon.engineeringserver.client.
DirectKAONConnection" )
```

In addition to `KAON_CONNECTION` further parameters, like user name or password might be necessary depending on the implementation and the type of connection to be used.

Create New OI-Model

Once a KAONConnection object has been obtained it can be used to create a new OI-Model (cf. listing A.2).

```
String m_sPhysicalURI = "file:/f:/temp/myTestOntology.kaon";
String m_sLogicalURI = "http://www.dummyurl.de/testOntology";
OIModel oimodel = connection.createOIModel( m_sPhysicalURI,
    m_sLogicalURI );
```

Listing A.2: Create new OI-Model

Each OI-Model is uniquely identified by two URIs - a physical and a logical one, which are totally independent from each other.

Physical URI The structure of the physical URI, which is used to access the OI-Model, depends on the KAON API implementation used. If an OI-Model is locally stored in `f:\temp\myTestOntology.kaon`, for instance, its physical URI would be `file:/f:/temp/myTestOntology.kaon`.

Logical URI A logical URI can be chosen freely, but in contrast to the physical URI it has to be globally unique. For example, the above mentioned OI-Model could have

the logical URI `http://www.dummyurl.de/testOntology`, although this URI does not really exist on the web.

Open OI-Model

An existing OI-Model can be opened by `connection.openOIModelPhysical( m_sPhysicalURI )`.

The method `connection.openOIModelLogical` only works for some well-known models (e.g. the lexical model) pre-registered with KAONConnection and for OI-Models, which are known to KAONConnection, because they have been previously opened by their physical URIs.

Add New Concepts

The following code fragment (cf. listing A.3) creates two new concepts, *Person* and *Document*.

```
Concept person =
    oimodel.getConcept( m_sLogicalURI +"#Person" );
Concept document =
    oimodel.getConcept( m_sLogicalURI +"#Document" );
```

Listing A.3: Create new concepts

Since the method `OIModel.getConcept` always returns a concept (even if there is no concept with the specified URI in the OI-Model), it can be used for both creating new concepts and accessing existing ones. The only parameter required by `OIModel.getConcept` is a unique URI for the concept to be created or retrieved. Very often, the logical URI of the OI-Model is used as a prefix for newly created concepts, because in this case the URIs of all model entities will be serialized relative to the model's URI, when the OI-Model is serialized in RDF.

As soon as the two new concepts have been created they can be added to the OI-Model as shown below:

```
List changes = new LinkedList();
changes.add( new AddEntity( person ) );
changes.add( new AddEntity( document ) );
changes.add( new AddSubConcept( oimodel.getRootConcept(),
    person ) );
changes.add( new AddSubConcept( oimodel.getRootConcept(),
    document ) );
oimodel.applyChanges( changes );
changes.clear();
```

Listing A.4: Add new concepts

With regards to ontology evolution (cf. subsection A.2) the KAON API does not allow for performing changes directly on the OI-Model. Therefore a list of change events has to be created and applied to the model[4]. In this case for each concept two change events are required: one for adding it to the OIModel and one for making it a subconcept of `root`.

Listing A.5 shows how to add a subconcept *Researcher* to the previously created concept *Person* and a subconcept *Paper* to *Document*.

```
Concept researcher =
    oimodel.getConcept( m_sLogicalURI +"#Researcher" );
Concept paper =
    oimodel.getConcept( m_sLogicalURI +"#Paper" );
changes.add( new AddEntity( researcher ) );
changes.add( new AddEntity( paper ) );
changes.add( new AddSubConcept( person, researcher ) );
changes.add( new AddSubConcept( document, paper ) );
oimodel.applyChanges( changes );
changes.clear();
```

Listing A.5: Add new subconcepts

It is important to know that **each new subconcept has to be added to the OI-Model (`AddEntity`)** before it can be made a subconcept of any other concept (`AddSubConcept`).

Add New Properties

Analogously to concepts (`OIModel.getConcept`) new properties can be created by `OIModel.getProperty`. The following code creates a new property *age* as an attribute to *Person*, a property *hasWritten(Researcher,Paper)* and a property *hasAuthor(Paper,Researcher)*.

```
Property age = oimodel.getProperty( m_sLogicalURI +"#age" );
Property hasWritten =
    oimodel.getProperty( m_sLogicalURI +"#hasWritten" );
Property hasAuthor =
    oimodel.getProperty( m_sLogicalURI +"#hasAuthor" );
changes.add( new AddEntity( age ) );
changes.add( new AddEntity( hasWritten ) );
changes.add( new AddEntity( hasAuthor ) );
changes.add( new AddPropertyDomain( age, person ) );
changes.add( new AddPropertyDomain( hasWritten, researcher ) );
changes.add( new AddPropertyDomain( hasAuthor, paper ) );
changes.add( new AddPropertyRange( hasWritten, paper ) );
changes.add( new AddPropertyRange( hasAuthor, researcher ) );
```

[4] All available change events are located in the `edu.unika.aifb.kaon.api.change` package.

```
oimodel.applyChanges( changes );
changes.clear();
```

Listing A.6: Add new properties

As shown by listing A.6 up to four steps are required for adding a new property to the OI-Model:

- Creating a property (`OIModel.getProperty`)
- Inserting the property into the OI-Model (`AddEntity`)
- Defining the domain of the property (`AddPropertyDomain`)
- Defining the range of the property (`AddPropertyRange`)

Subproperties can be created by using the change event `AddSubProperty( superProperty, subProperty )` (cf. subsection A.1).

Instantiate Concepts

Since an OI-Modelmay not only include concepts and properties, but also instances of both, the KAON API provides methods for the instantiation of concepts (cf. listing A.7) and properties (cf. subsection A.2).

The following example demonstrates how instances of the concepts *Researcher* and *Paper* can be added to the OI-Model.

```
Instance erwin =
    oimodel.getInstance( m_sLogicalURI +"#Erwin_Skela" );
Instance david =
    oimodel.getInstance( m_sLogicalURI +"#David_Montero" );
Instance paperXYZ =
    oimodel.getInstance( m_sLogicalURI +"#Paper_XYZ" );
changes.add( new AddInstanceOf( researcher, erwin ) );
changes.add( new AddInstanceOf( researcher, david ) );
changes.add( new AddInstanceOf( paper, paperXYZ ) );
oimodel.applyChanges( changes );
changes.clear();
```

Listing A.7: Add new instances

Instantiate Properties

The instantiation of properties is very similar to the instantiation of concepts described in the previous subsection. The code fragment shown by listing A.8 instantiates the properties *age* and *hasWritten(Researcher,Paper)* by assigning an age of 27 to *Erwin_Skela* and creating a *hasWritten* relation between *David_Montero* and *Paper_XYZ*.

```
PropertyInstance erwin_age_27 =
    oimodel.getPropertyInstance( age, erwin, "27" );
PropertyInstance david_hasWritten_paperXYZ =
    oimodel.getPropertyInstance( hasWritten, david, paperXYZ );
changes.add(
    new AddPropertyInstance( erwin_age_27 ) );
changes.add(
    new AddPropertyInstance( david_hasWritten_paperXYZ ) );
oimodel.applyChanges( changes );
changes.clear();
```

Listing A.8: Instantiate properties

Pose Queries

KAON Query is an experimental conceptual query language, which is part of the KAON ToolSuite (cf. figure 4.1). It can be used programmatically by means of the KAON API for efficient locating of OI-Model elements. The following example demonstrates how to retrieve all instances of the concept *Paper*, which has been created in subsection A.1.

```
String sQuery = "["+ m_sLogicalURI +"#Paper]";
Collection answer = oimodel.executeQuery( sQuery );
```

Listing A.9: Pose queries

Since *Paper_XYZ* is the only instance of *Paper*, the collection returned by `OIModel.executeQuery` in the above listed code fragment contains only one element (cf. listing A.10).

```
Query: [http://www.dummyurl.de/testOntology#Paper]
Answer: http://www.dummyurl.de/testOntology#Paper_XYZ
```

Listing A.10: Query result

Since all instances of a certain concept can also be retrieved by using the `Concept.getInstances()` method, the same result would be returned by `paper.getInstances()`.

Remove Concepts

As shown by listing A.11 a concept can be removed from an OI-Model by using the `RemoveEntity` change event.

```
changes.add( new RemoveEntity( paper ) );
List requestedChanges =
    oimodel.applyChanges( requestedChanges );
changes.clear();
```

Listing A.11: Remove a concept

Similar change events are provided for removing subconcepts, properties or instances, for example.

The impact of removing concepts or other entities such as properties or instances from an OI-Model is demonstrated by the following RDF serialization (cf. subsection A.2) of the sample ontology created in the previous subsections.

```
<rdf:RDF xml:base="http://www.dummyurl.de/testOntology"
    xmlns:rdf="&rdf;"
    xmlns:rdfs="&rdfs;"
    xmlns:a="&a;">
<a:Researcher rdf:ID="David_Montero"/>
<rdfs:Class rdf:ID="Document"/>
<a:Researcher rdf:ID="Erwin_Skela" a:age="27"/>
<rdfs:Class rdf:ID="Person"/>
<rdfs:Class rdf:ID="Researcher">
    <rdfs:subClassOf rdf:resource="#Person"/>
</rdfs:Class>
<rdf:Property rdf:ID="age">
    <rdfs:domain rdf:resource="#Person"/>
</rdf:Property>
</rdf:RDF>
```

Listing A.12: Remove concept *Paper*

Obviously, the deletion of the concept *Paper* entails the deletion of

- its instance *Paper_XYZ*,
- the property *hasWritten(Researcher,Paper)*,
- the property *hasAuthor(Paper,Researcher)*,
- and the property instance *hasWritten(David_Montero,Paper_XYZ)*.

In order to manage such complex changes and to avoid potential problems like inconsistencies, for example, KAON provides different evolution strategies. The next subsection describes how these evolution strategies can be employed by means of the KAON API.

Use Evolution Strategies

Since each change to an OI-Model might potentially cause inconsistencies in this as well as in dependent ontologies, the KAON API supports the use of different evolution strategies (cf. [SMMS02b] and [SSH02])). The following code fragment below shows how to use evolution strategies considering as example the deletion of *Paper* described in the previous subsection.

```
EvolutionStrategy strategy =
    new EvolutionStrategyImpl( oimodel );
changes.add( new RemoveEntity( paper ) );
List requestedChanges =
    strategy.computeRequestedChanges( changes );
oimodel.applyChanges( requestedChanges );
changes.clear();
```

Listing A.13: Evolution strategies

Once an EvolutionStrategy object has been created it can be used in order to transform a list of change events (e.g. `RemoveEntity(Paper)`) into a new list containing all the change events, which are necessary for safely performing the desired changes. The content as well as the sequential order of this list depends on the evolution strategy implementation and the evolution parameters (`edu.unika.aifb.kaon.defaultevolution.EvolutionParameters`) specified by the user.

Serialization

The easiest way of serializing an OI-Model is to apply the `OIModel.save()` method, which stores the OI-Model either to a local file (determined by its physical URI) or, for instance, to a database - depending on which implementation of the KAON API is currently used.

Nevertheless, for debugging purposes it might be useful to create a textual output of the OI-Model . In this case the `RDFSerializer` class can be instantiated in order to write an RDF serialization to an output stream such as System.out (cf. listing A.14).

```
RDFSerializer serializer = RDFManager.createSerializer();
serializer.serialize( ((OIModelImpl)oimodel).getModel(),
    System.out, "UTF-8" );
writer.close();
```

Listing A.14: RDFSerializer

The following extract shows an RDF serialization of the sample ontology, which has been created in the subsections A.2 to A.2.

```
<rdf:RDF xml:base="http://www.dummyurl.de/testOntology"
    xmlns:rdf="&rdf;"
    xmlns:rdfs="&rdfs;"
    xmlns:a="&a;">
<a:Researcher rdf:ID="David_Montero">
    <a:hasWritten rdf:resource="#Paper_XYZ"/>
</a:Researcher>
<rdfs:Class rdf:ID="Document"/>
<a:Researcher rdf:ID="Erwin_Skela" a:age="27"/>
```

```
<rdfs:Class rdf:ID="Paper">
    <rdfs:subClassOf rdf:resource="#Document"/>
</rdfs:Class>
<a:Paper rdf:ID="Paper_XYZ"/>
<rdfs:Class rdf:ID="Person"/>
<rdfs:Class rdf:ID="Researcher">
    <rdfs:subClassOf rdf:resource="#Person"/>
</rdfs:Class>
<rdf:Property rdf:ID="age">
    <rdfs:domain rdf:resource="#Person"/>
</rdf:Property>
<rdf:Property rdf:ID="hasAuthor">
    <rdfs:domain rdf:resource="#Paper"/>
    <rdfs:range rdf:resource="#Researcher"/>
</rdf:Property>
<rdf:Property rdf:ID="hasWritten">
    <rdfs:domain rdf:resource="#Researcher"/>
    <rdfs:range rdf:resource="#Paper"/>
</rdf:Property>
</rdf:RDF>
```

Listing A.15: RDF serialization

A.3 KAON Engineering Server

Currently, there are three different back-end implementations of the KAON API (cf. Section A.2): the **Main Memory** implementation, the **RDF Server** as well as the **Engineering Server**. In this chapter we focus on the latter, most sophisticated KAON implementation, the Engineering Server.

Motivation

When building large ontologies (with probably thousands of concepts and relations and maybe millions of instances), it is with current standard technology rather infeasible to store that amount of data in main memory. In fact, when the ontology to be built exceeds a certain size, the usage of a database management system storing the mass of data is unavoidable. For that purpose, i.e. for managing the interaction with the database system, KAON includes an implementation of the so-called Engineering Server fulfilling that task.

In short, the Engineering Server is a **storage mechanism for KAON ontologies, based on relational databases and suitable for use during ontology engineering**. Its features include

- transactions,

- client-side caching with conflict detection,
- distributed change notification mechanism,
- bulk-loading of ontology elements,
- modularization (limited to models within the same database).

The Engineering Server has been tested with an ontology consisting of 100.000 concepts, 66.000 properties and 1.000.000 instances, where loading related information about 20 ontology entities took under 3 seconds, while deleting a concept took under 5 seconds.

Database Access

The Engineering Server is an ontology server that is optimized for ontology engineering. This optimization is in particular reflected in the database schema used by the server. Since ontology engineering often involves creating and deleting concepts, which should be multi-user capable and transactional, the engineering server has a database schema with a *fixed* number of tables.

An obvious alternative realization of an ontology servers might create a table per concept. However, this would make concept creation and deletion non-transactional, and in general, more heavy-weight. The schema employed by the Engineering Server is presented in Figure 1: One can see that it consists of a fixed number of tables. Indeed, this fact distinguishes the Engineering Server from other ontology servers implementations, which store all instances of a concept in a separate table (and thus require table creation and deletion every time a concept is created).

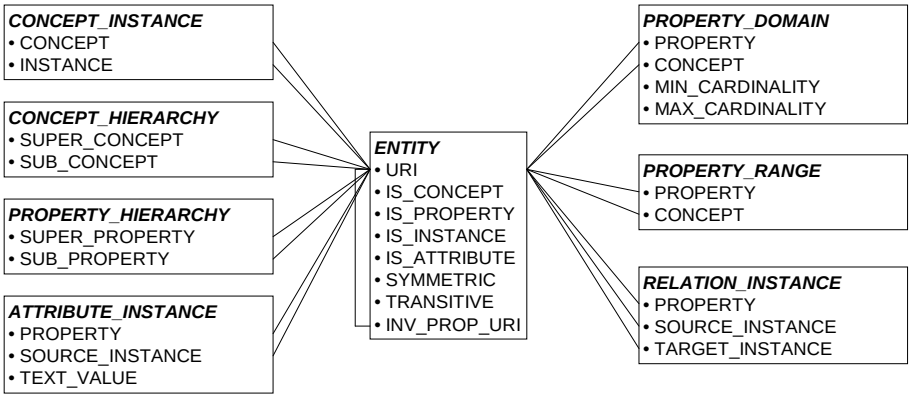

Figure A.18: Engineering Server Database Schema

The Engineering Server offers scalable, transactional and concurrent access to ontology information. To achieve that, optimized bulk-loading is implemented, that allows fetching information about several ontology entities in one database request.

The Engineering Server has been tested with MSL SQL Server 2000. Also, it has successfully been run, but not thoroughly tested, on PostgreSQL, IBM DB2 7.2 and Oracle 9i. Although other databases may work, the Hypersonic database will *not* work, since it doesn't support many of standard features of relational databases that are needed for server operation. For users of IBM DB2 it is important to configure the DB2 database to use JDBC2 – please refer to the DB2 documentation for information on how to perform that.

Usage Scenario for the Engineering Server

Mainly, there are three ways of using the Engineering Server (Direct, Remote, and Local) which we describe in more detail in the subsequent sections.

In general, clients (e.g. ontology editors) access the Engineering Server through an appropriate KAON API implementation (e.g. the API Proxy, cf. Figure 4.1), which provides client-side ontology information caching, along with a mechanism for detecting incoherencies between the cache and the database. This feature thus significantly simplifies developing applications where ontology entities are loaded and kept at the client across transaction boundaries.

Before using the Engineering Server, it is necessary to create the necessary schema in the database used. For this, the Engineering Server comes with a number of scripts (SQL scripts, database-specific) whose execution results in the creation of the necessary database tables. If those scripts can be executed without errors, the Engineering Server is ready for usage. For more details concerning obtaining and installing the Engineering Server please refer to Appendix A.4.

Direct Engineering Server The *Direct Engineering Server* corresponds to a *two-tiered* setting (database and Engineering Server). This means that the Engineering Server accesses the database directly, which, of course, may be played on another host. In this setting distributed change notification is not available.

The Direct Engineering Server is useful for any type of application in which distributed event notification is not required. So, it represents the simplest variant to set up and use the Engineering Server (because it does not require a J2EE server such as JBoss).

Remote/Local Engineering Server In this *three-tiered* setting an additional intervening application server JBoss[5])is employed. The main advantage of these forms of the Engineering Server is that clients can register themselves to be notified whenever other users make a change in the ontology. Thus the Engineering Server can serve as a basis for collaborative ontology development.

In the case of using the *Remote Engineering Server* that JBoss Application Server is accessed through another Java Virtual Machine (JVM) through remote interfaces. The

[5] `http://www.jboss.org`

Remote Engineering Server is useful for applications which need distributed event notification. This in particular relates to applications where ontologies are manipulated by several users concurrently, such as an ontology editor.

The *Local Engineering Server* represents a *three-tiered* solution as well. Here, however, the Engineering Server accesses the JBoss application Server from within the same JVM through local interfaces. The Local Engineering Server is useful in particular for web applications, since the web application and the server can be deployed in the same JVM, and thus increase performance (since the remote call overhead is eliminated).

Both, the Remote and Local Engineering Servers come in two versions: secure and non-secure. In the non-secure version no authentication of clients, that want to access the ontology/database, is needed. For the secure version, authentication is realized via JBoss. Note, that the version of the Engineering Server being used (i.e. secure or non-secure) is determined at deployment time.

Collaborative Ontology Engineering with the Engineering Server

As emphasized before, the main benefit of using the Engineering Server—apart from handling huge amounts of data via accessing a relational database system—is the possibility to collaboratively work on a single ontology model instance. That individual instance is maintained by the Engineering Server to which clients may connect.

Due to these multi-user capabilities of the Engineering Server it is feasible to develop an ontology in a distributed setting, e.g. with a group of ontology engineers or domain experts who are spread across several locations. Here, each participant connects to the instance of the Engineering Server (e.g. running on a server in Karlsruhe) with his/her local client (e.g. the OI-Modeler as part of the KAON Workbench). Then, it is possible to browse and explore the entire ontology without restrictions. Furthermore, if the respective user has the appropriate rights for writing, i.e. is also allowed to apply changes to the ontology, he/she may add, change, or delete ontology entities.

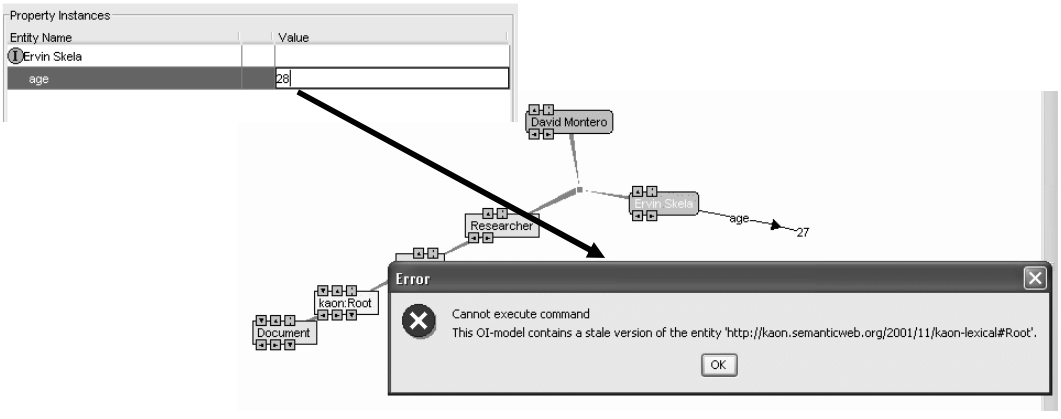

Figure A.19: Ontology Version Conflict during Distributed Ontology Engineering

At its current stage of development the Engineering Server assigns a version number to each successive state of the ontology. In case a client's local "copy" of the ontology has a lower version number than the current ontology version maintained by the Engineering Server, the user is prompted that a conflict exists.

Imagine ontology engineer A has noticed that `Erwin Skela` was misspelled and corrects that typo so that that instance is called `Ervin Skela`. Moreover, ontology engineer B wants to change that instance's value for attribute AGE from 27 to 28. In case B has not updated his/her OI-Model, he/she will be notified about the conflict as depicted in Figure A.19: Obviously, B is not allowed to change the value for attribute AGE as his/her current version of the ontology is obsolete. Now, ontology engineer B would have to refresh his/her ontology—which means that the Engineering Server's current ontology version is transferred to the client—and must reapply the changes he/she wanted to introduce, i.e. change the value for the AGE attribute as desired.

A.4 Download & Installation

This chapter gives hints concerning download and installation of tools and components related to KAON.

After a short overview on current versions and download sites we devote a single subsection to the installation of each tool presented in this document.

Download Overview

The following table A.1 summarizes download sources for and version numbers of the tools described within this document. Note, that future versions might not necessarily be compatible with the current versions described here. Be aware, that KAON needs at least Java 1.4.0.

Tool	**Version**	**Download Site**
KAON	V1.2.7	http://sourceforge.net/projects/kaon
Note: You may choose between the source code and a binary version.		
KAON Extensions *Includes:* KAONToEdit	V1.0	http://sourceforge.net/projects/kaon-ext/
TextToOnto	V0.95b	http://sourceforge.net/projects/texttoonto/
Java	V1.4.2	http://java.sun.com/j2se/1.4.2/download.html
OntoEdit	V2.6	http://www.ontoprise.de/customercenter/software_downloads/
JBoss	V3.2.1	http://www.jboss.org

Table A.1: Downloading KAON

Installation of KAON, its Workbench, and OI-Modeler

After having obtained KAON from the download site mentioned in Table A.1, you may proceed making that software applicable.

Note, that the term "KAON Workbench" (cf. Section 4.1) in particular comprises the OI-Modeler, KAON's ontology editor.

Installation: Installing KAON and its Workbench is straightforward. To install KAON just unpack the downloaded archive to some directory without spaces.

Using OI-Modeler: Set the KAON_ROOT environment variable to point to the root of your KAON distribution, for example via `c:\>set KAON_ROOT=c:\kaon`. Then, you may start the OI-Modeler by invoking `KAON_ROOT\bin\kaongui.bat`

As shown in Figure 4.1 the KAON archive you have downloaded comprises KAON's Engineering Server as well. However, since installing and using that software involves a number of steps to be followed, we describe that proceeding in very detail in the following section.

Installation of the Engineering Server

The Engineering Server represents KAON's implementation for large-scale and distributed ontology engineering. Note, that the Engineering Server is a part of KAON. So, the files and libraries related to it are included in the KAON archive you probably have downloaded and installed in Section A.4. The following steps describe how to install and use the Engineering Server.

Exemplarily, we describe the installation process for Microsoft SQL Server 2000 as the database system the Engineering Server collaborates with, for which it has been tested thoroughly. However, it should work in principle with all SQL2-compatible databases. It has been successfully run on IBM DB2, PostgreSQL and Oracle 8i/9i.

Direct Engineering Server

Applying the Direct Engineering Server corresponds to a two-tiered setting. Of course, the functionality of both tiers (i.e. client and relational database system) may be placed on the same machine. Anyway, the first phase of the Engineering Server's installation involves the creation of the server's database and filling it with the necessary database schema (see Section A.18).

1. Install the relational database system and make sure it is up and running.

2. Create a new database (e.g. `myOntologyDatabase`) with your database management tool (in case of MS SQL Server that tool is the "SQL Server Enterprise Manager").

3. Use your database tool to execute the `schema.sql` script located in `KAON_ROOT\engineeringserver\schema\`. For example, for the MS SQL Server you may use the "SQL Query Analyzer" for that purpose. That script will create the schema in your database. The file is in the SQL2 format and uses the semicolon character as the command separator. Depending on your database, you may need to replace that character with the keyword used by your database (for example, *older* versions of MS SQL Server used the keyword GO as the command separator).

4. Depending on your database (i.e. if you are not using MS SQL Server), execute the supplementary schema script. For example, in case of Oracle database, execute `engineeringserver\schema\schema_oracle.sql` file. The same comments about the command separator apply.

5. Create a database user with your database management tool, e.g. MS SQL Server Enterprise Manager. Pay also attention that the security settings of MS SQL Server Enterprise Manager do not only allow for Windows authentication, but for "SQL Server and Windows".

Now, the Engineering Server is ready to be used in its *direct* version. From within the OI-Modeler you may create a new or open an existing ontology model via the Direct Engineering Server by supplying the following information:

- Start the OI-Modeler and choose open or create an OI-Model.
- Choose the tab "Direct Engineering Server".
- User Name and Password as specified in 5.
- Host Name: localhost or the server the SQL Server is running on
- Driver: Here, you can choose between MS SQL Server, IBM DB2, Oracle, PostgreSQL, and other.
- Port: 1433
- Database: name of your database, e.g. `myOntologyDatabase`

Remote/Local Engineering Server

For the *remote or local* version of the Engineering Server, you need the JBoss application server and you have to deploy the Engineering Server's EJBs to that application server. To do so, follow these steps:

1. Download JBoss from `http://www.jboss.org/downloads.jsp`. Unpack the JBoss into a directory without spaces.

2. Set the `JBOSS_HOME` environment variable to point to the root of the JBoss distribution (e.g. `C:\java\jboss-3.2.1_tomcat-4.1.24`).

3. KAON distribution contains JBoss 3.2.1 libraries (in the 3rdparty directory) that facilitate connection to the JBoss application server. The version of the client-side libraries must match to the JBoss version. If you are using some other version of JBoss, then you should exchange the JBoss libraries in the 3rdparty directory with the appropriate libraries from your JBoss distribution.

   Please note, that we conducted all our tests with version 3.2.1 of JBoss, thus we recommend using that JBoss version as incompatibilities might arise otherwise.

4. Customize JBoss to connect to your database. This involves the following steps:

   - Copy the template database configuration file for your database (for MS SQL Server that file is called `mssql-ds.xml`) from `JBOSS_HOME\docs\examples\jca` directory to `JBOSS_HOME\server\default\deploy`.
   - Open that file in a text editor.
   - Customize the name of the JNDI data source to KAON (by editing the value of the `<jndi-name>` element).
   - Enter other information about your database (such as connection string, user name and password).
   - Make the JDBC driver available to JBoss by copying it to `JBOSS_HOME\server\default\lib` directory. The JDBC driver for MS SQL, for example, consists of three files (`msbase.jar`, `mssqlserver.jar`, and `msutil.jar`).

5. Start JBoss (by invoking the `JBOSS_HOME\bin\run.bat` script).

6. Deploy the Engineering Server. Here, you will have to decide whether you intend to use the Engineering Server in its *secure* or *non-secure* version (see below). For the non-secure version invoke the `engineeringserver\deploy.bat` script (which will copy `engineeringserver-beans.jar` file to the `JBOSS_HOME\server\default\deploy` directory). For the secure version invoke the `deploy_secure.bat` script (which will copy `engineeringserversecure-beans.jar` file to `JBOSS_HOME\server\default\deploy` directory).

Now, the Local/Remote Engineering Server is ready for use. From within the ontology editor OI-Modeler you may now access it. Depending on the decision whether you intend to use the Engineering Server in its non-secure or secure version, proceed as follows:

Non-Secure Version In the non-secure version no authentication mechanism is present.

- Start the OI-Modeler and choose open or create an OI-Model.
- Choose the tab “Engineering Server”.
- Host Name: localhost or the server on which JBoss is running
- Port: 1099
- User Name and Password: You can leave these fields blank as no authentication methods are employed in the non-secure version.

Secure Version In the secure version the same information as in the non-secure version have to be supplied. Moreover, you have to fill in your user name and password allowing you to access JBoss. Of course, you have to customize JBoss in prior so that it supports authentication. This basically means, you have to define a set of users and grant them rights to read and/or modify the ontology.

For this you must set up a security domain called `kaon` and specify how authentication is performed. A simple way of authenticating users is by using `UsersRolesLoginModule` of JBoss. This module expects two files in `JBOSS_HOME\server\default\conf` directory: `users.properties` contains entries of the form `userName=password`, whereas `roles.properties` contains entries of the form `userName=role1,role2`.

The Engineering Server supports two roles: `KAONReader` role allows the user to read the OI-Model, whereas `KAONWriter` role allows user to change the OI-Model. The `UsersRolesLoginModule` can be started by editing `JBOSS_HOME\server\default\conf\login-config.xml` file and appending the following fragment:

```
<application-policy name="kaon">
 <authentication>
  <login-module
   code=
   "org.jboss.security.auth.spi.UsersRolesLoginModule"
   flag="required" />
 </authentication>
</application-policy>
```

For installation of other authentications modules, please refer to JBoss documentation.

Installation of TextToOnto

TextToOnto has been developed as an open-source project and therefore can be freely obtained from the address mentioned in table A.1. Because it does not need any additional

libraries or software apart from the Java Runtime Environment[6], the installing and running TextToOnto is straightforward:

- decompress the binary distribution into a directory <`INST-DIR`> (for example `c:\TextToOnto`)
- go to <`INST-DIR`>`\bin`
- set the classpath and start TextToOnto:

  DOS / Windows: invoke <`INST-DIR`>`\bin\texttoonto.bat`

  Unix / Linux:

  – execute the shell script `../sentenv.sh` in order to set up your environment

  – start TextToOnto via `java -cp "%TEXTTOONTO_CLASSPATH%" edu.unika.aifb.texttoonto.TextToOnto`

Moreover, although this is not required for using TextToOnto, you might want to install WordNet 1.7.1 which significantly improves the results of the TaxoBuilder module. WordNet distributions for various operating systems can be downloaded from `http://www.cogsci.princeton.edu/~wn/`.

[6] http://java.sun.com/j2se/desktopjava/jre/index.jsp

Bibliography

[AFM03] Alessandro Artale, Enrico Franconi, and Federica Mandreoli. Description logics for modeling dynamic information. In *Logics for Emerging Applications of Databases*, 2003.

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

[BHGS01] S. Bechhofer, I. Horrocks, C. Goble, and R. Stevens. OilEd: A reason-able ontology editor for the semantic web. In *KI-2001: Advances in Artificial Intelligence*, LNAI 2174, pages 396–408. Springer, 2001.

[BKKK87] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 311–322. ACM Press, 1987.

[BLHL01] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 2001(5), 2001. available at http://www.sciam.com/2001/0501issue/0501berners-lee.html.

[BvHH+] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. Owl web ontology language reference. `http://www.w3.org/TR/owl-ref/`.

[C+03] Per Cederqvist et al. *The CVS manual - Version Management with CVS*. Network Theory Limited, 2003.

[CFF+98] V. K. Chaudhri, A. Farquhar, R. Fikes, P. D. Karp, and J. Rice. OKBC: A programmatic foundation for knowledge base interoperability. In *AAAI/IAAI*, pages 600–607, 1998.

[CST03] P. Cimiano, S.Staab, and J. Tane. Automatic acquisition of taxonomies from text: Fca meets nlp. In *Proceedings of the PKDD/ECML'03 International Workshop on Adaptive Text Extraction and Mining*, 2003.

[DP04] Zhongli Ding and Yun Peng. A Probabilistic Extension to Ontology Language OWL. In *Proceedings of the 37th Hawaii International Conference On System Sciences (HICSS-37).*, Big Island, Hawaii, January 2004.

[EGH+04a] M. Ehrig, T. Gabel, P. Haase, Y. Sure, C. Tempich, and J. Voelker. Data manual - initial version. SEKT informal deliverable 7.1.1.b, Institute AIFB, University of Karlsruhe, 2004.

[EGH+04b] M. Ehrig, T. Gabel, P. Haase, Y. Sure, C. Tempich, and J. Voelker. Use cases - initial version. SEKT informal deliverable 7.1.1.a, Institute AIFB, University of Karlsruhe, 2004.

[Fel98] C. Fellbaum. *WordNet, an electronic lexical database*. MIT Press, 1998.

[FGM00] Enrico Franconi, Fabio Grandi, and Federica Mandreoli. A semantic approach for schema evolution and versioning in object-oriented databases. In *Proceedings of the First International Conference on Computational Logic*, pages 1048–1062. Springer-Verlag, 2000.

[GPAFL+02] A. Gómez-Pérez, J. Angele, M. Fernandéz-López, V. Christophides, A. Stutt, Y. Sure, et al. A survey on ontology tools. OntoWeb deliverable 1.3, Universidad Politecnia de Madrid, 2002.

[GSV04] T. Gabel, Y. Sure, and J. Voelker. KAON – ontology management infrastructure. SEKT informal deliverable 3.1.1.a, Institute AIFB, University of Karlsruhe, 2004.

[GW99] B. Ganter and R. Wille. *Formal Concept Analysis. Mathematical Foundations*. Springer Verlag, 1999.

[Hea92] M.A. Hearst. Automatic acquisition of hyponyms from large text corpora. In *Proceedings of the 14th International Conference on Computational Linguistics*, 1992.

[HEHS04] P. Haase, M. Ehrig, A. Hotho, and B. Schnizler. Personalized information access in a bibliographic peer-to-peer system. In *Proceedings of the AAAI Workshop on Semantic Web Personalization, 2004*, JUL 2004.

[HH00] Jeff Heflin and James A. Hendler. Dynamic ontologies on the web. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 443–449. AAAI Press / The MIT Press, 2000.

[HH02] I. Horrocks and J. A. Hendler, editors. *Proceedings of the First International Semantic Web Conference: The Semantic Web (ISWC 2002)*, volume 2342 of *Lecture Notes in Computer Science (LNCS)*, Sardinia, Italy, 2002. Springer.

[HMS04a] U. Hustadt, B. Motik, and U. Sattler. Reasoning for Description Logics around $\mathcal{SHIQ}$ in a Resolution Framework. Technical Report 3-8-04/04, FZI, Karlsruhe, Germany, April 2004. http://www.fzi.de/wim/publikationen.php?id=1172.

[HMS04b] U. Hustadt, B. Motik, and U. Sattler. Reducing $\mathcal{SHIQ}^-$ Description Logic to Disjunctive Datalog Programs. In D. Dubois, C. Welty, and M.-A. Williams, editors, *Proc. of the 9th Int. Conf. on Knowledge Representation and Reasoning (KR2004)*, pages 152–162, Menlo Park, California, USA, June 2004. AAAI Press.

[HPS04] I. Horrocks and P. F. Patel-Schneider. Reducing OWL Entailment to Description Logic Satisfiability. *Journal of Web Semantics*, 1(4), 2004.

[HPSvH03] I. Horrocks, P. F. Patel-Schneider, and F. van Harmelen. From SHIQ and RDF to OWL: The Making of a Web Ontology Language . *Journal of Web Semantics*, 1(1), 2003.

[HS98] U. Hahn and K. Schnattinger. Towards text knowledge engineering. In *AAAI'98/IAAI'98 Proceedings of the 15th National Conference on Artificial Intelligence and the 10th Conference on Innovative Applications of Artificial Intelligence*, 1998.

[HST00] I. Horrocks, U. Sattler, and S. Tobies. Practical Reasoning for Very Expressive Description Logics. *Logic Journal of the IGPL*, 8(3):239–263, 2000.

[HV04] P. Haase and J. Voelker. Requirements analysis for usage-driven and data-driven change discovery. SEKT informal deliverable 3.3.1.a, Institute AIFB, University of Karlsruhe, 2004.

[Kar02] FZI Karlsruhe. OI-Modeler user's guide, 2002.

[KF01] Michel Klein and Dieter Fensel. Ontology versioning for the Semantic Web. In *Proceedings of the First International Semantic Web Working Symposium (SWWS)*, pages 75–91, Stanford University, California, USA, July 30 – August 1, 2001.

[KFKO02] Michel Klein, Dieter Fensel, Atanas Kiryakov, and Damyan Ognyanov. Ontology versioning and change detection on the web. In *13th International Conference on Knowledge Engineering and Knowledge Management (EKAW02)*, number 2473 in LNCS, page 197 ff, Sigüenza, Spain, October 1–4, 2002.

[KK04] FZI Karlsruhe and AIFB Karlsruhe. KAON the Karlsruhe ontology and semantic web framework — developer's guide for KAON 1.2.7, 2004.

[KKOF02] Michel Klein, Atanas Kiryakov, Damyan Ognyanov, and Dieter Fensel. Finding and characterizing changes in ontologies. In *Proceedings of the 21st International Conference on Conceptual Modeling (ER2002)*, number 2503 in LNCS, pages 79–89, Tampere, Finland, October 7–11, 2002.

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

[KN03] Michel Klein and Natalya F. Noy. A component-based framework for ontology evolution. In *Proceedings of the Workshop on Ontologies and Distributed Systemsm, IJCAI '03*, Acapulco, Mexico, August9, 2003. Also available as Technical Report IR-504, Vrije Universiteit Amsterdam.

[MD00] T.J. Menzies and J. Debenham. Expert systems maintenance. *Encyclopedia of Computer Science and Technology*, 47(27):35–54, 2000. Available from `http://tim.menzies.com/pdf/00cst.pdf`.

[Men99] Tim Menzies. Knowledge maintenance: The state of the art. *The Knowledge Engineering Review*, 14(1), 1999.

[MM04] F. Manola and E. Miller. Resource Description Framework (RDF). primer. W3C Recommendation 10 February 2004, 2004. available at http://www.w3.org/TR/rdf-primer/.

[MMS03a] Alexander Maedche, Boris Motik, and Ljiljana Stojanovic. Managing multiple and distributed ontologies in the semantic web. *VLDB Journal*, 12(4):286–302, 2003.

[MMS+03b] A. Maedche, B. Motik, L. Stojanovic, R. Studer, and R. Volz. An infrastructure for searching, reusing and evolving distributed ontologies. In *Proceedings of the twelfth international conference on World Wide Web*, pages 439–448, Budapest, Hungary, 2003. ACM Press.

[MMV02] A. Maedche, B. Motik, and R. Volz. A conceptual modeling approach for semantics-driven enterprise applications. In Springer-Verlag, editor, *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBA*, pages 1082 – 1099, October 30 - November 01 2002.

[MP02] Peter McBrien and Alexandra Poulovassilis. Schema evolution in heterogeneous database architectures, a schema transformation approach. In *Proceedings of the 14th International Conference on Advanced Information Systems Engineering*, pages 484–499. Springer-Verlag, 2002.

[MS00] A. Maedche and S. Staab. Discovering conceptual relations from text. In W. Horn, editor, *Proceedings of the 14th European Conference on Artificial Intellignece (ECAI'2000)*, 2000.

[MS02] A. Maedche and S. Staab. Measuring similarity between ontologies. In *Proceedings of the European Conference on Knowledge Acquisition and Management (EKAW)*. Springer, 2002.

[MSSV02] A. Maedche, L. Stojanovic, R. Studer, and R. Volz. Managing multiple ontologies and ontology evolution in ontologging. In *Proceedings of the IFIP 17th World Computer Congress – TC12 Stream on Intelligent Information Processing*, pages 51 – 63, Montreal, Canada, 2002. Kluwer.

[MV01] A. Maedche and R. Volz. The ontology extraction and maintenance framework text-to-onto. In *Proceedings of the ICDM'01 Workshop on Integrating Data Mining and Knowledge Management*, 2001.

[NFM00] N. Noy, R. Fergerson, and M. Musen. The knowledge model of Protégé-2000: Combining interoperability and flexibility. In R. Dieng and O. Corby, editors, *Proceedings of the 12th International Conference on Knowledge Engineering and Knowledge Management: Methods, Models, and Tools (EKAW 2000)*, volume 1937 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 17–32, Juan-les-Pins, France, 2000. Springer.

[NK03] Natalya F. Noy and Michel Klein. Ontology evolution: Not the same as schema evolution. *Knowledge and Information Systems*, 5, 2003. in press.

[oG03] ontoprise GmbH. How to work with OntoEdit — user's guide for OntoEdit version 2.6. http://www.ontoprise.de/documents/tutorial_ontoedit.pdf, September 2003.

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

[PK97] Anne Pons and Rudolf R. Keller. Schema evolution in object databases by catalogs. In *Proceedings of the International Database Engineering and Applications Symposium (IDEAS'97), Montreal, Canada*, pages 368–376, 1997.

[PSHH] Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks. Owl web ontology language semantics and abstract syntax. `http://www.w3.org/TR/2004/REC-owl-semantics-20040210/`.

[Pz97] Randel J. Peters and M. Tamer Özsu. An axiomatic model of dynamic schema evolution in objectbase systems. *ACM Trans. Database Syst.*, 22(1):75–114, 1997.

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

[Rou04] Marie-Christine Rousset. Small can be beautiful in the semantic web. In *Proceedings of the 3rd International Semantic Web Conference (ISWC 2004)*, Hiroshima, Japan, November 2004.

[SAS03] Y. Sure, J. Angele, and S. Staab. OntoEdit: Multifaceted inferencing for ontology engineering. *Journal on Data Semantics*, LNCS(2800):128–152, 2003.

[SASS04] L. Stojanovic, A. Abecker, N. Stojanovic, and R. Studer. On managing changes in the ontology-based e-government. In *Proceedings of the 3rd International Conference on Ontologies, Databases and Application of Semantics (ODBASE 2004)*, number 3291 in Lecture Notes in Computer Science, pages 1080–1097, Agia Napa, Cyprus, November 2004. Springer Verlag.

[SEA+02] Y. Sure, M. Erdmann, J. Angele, S. Staab, R. Studer, and D. Wenke. OntoEdit: Collaborative ontology development for the semantic web. In Horrocks and Hendler [HH02], pages 221–235.

[SHG03] N. Stojanovic, J. Hartmann, and J. Gonzalez. Ontomanager - a system for usage-based ontology management. In *In Proceedings of FGML Workshop. Special Interest Group of German Information Society (FGML - Fachgruppe Maschinelles Lernen der GI e.V.)*, 2003.

[SK03] Heiner Stuckenschmidt and Michel Klein. Integrity and change in modular ontologies. In *Proceedingso of the 18th International Joint Conference on Artificial Intelligence*, Acapulco, Mexico, August 2003.

[SMMS02a] Ljiljana Stojanovic, Alexander Mädche, Boris Motik, and Nenad Stojanovic. User-driven ontology evolution management. In *European Conf. Knowledge Eng. and Management (EKAW 2002)*, pages 285–300. Springer-Verlag, 2002.

[SMMS02b] L. Stojanovic, A. Maedche, B. Motik, and N. Stojanovic. User-driven ontology evolution management. In *Proceedings of the 13th European Conference on Knowledge Engineering and Knowledge Management EKAW*, volume 2473 of *Lecture Notes in Computer Science*, pages 285 – 300, Siguenza, Spain, October 1-4 2002. Springer.

[SMSS03] Ljiljana Stojanovic, Alexander Maedche, Nenad Stojanovic, and Rudi Studer. Ontology evolution as reconfiguration-design problem solving. In *KCAP 2003*, pages 162–171. ACM, OCT 2003.

[SR03] Ganesan Shankaranarayanan and Sudha Ram. Research issues in database schema evolution - the road not taken. Technical Report 2003-15, The University of Arizona, 2003.

[SSH02] L. Stojanovic, N. Stojanovic, and S. Handschuh. Evolution of the metadata in the ontology-based knowledge management systems. In *German Workshop on Experience Management*, pages 65 – 77, 2002.

[ST04] Y. Sure and C. Tempich. State of the art in ontology engineering methodologies. SEKT informal deliverable 7.1.2, Institute AIFB, University of Karlsruhe, 2004.

[Sto04a] Ljiljana Stojanovic. *Methods and Tools for Ontology Evolution*. PhD thesis, University of Karlsruhe, 2004.

[Sto04b] Ljlijana Stojanovic. *Methods and Tools for Ontology Evolution*. PhD thesis, University of Karlsruhe, 2004.

[SWM04] M. K. Smith, C. Welty, and D. McGuinness. OWL Web Ontology Language Guide, 2004. W3C Recommendation 10 February 2004, available at http://www.w3.org/TR/owl-guide/.

[TKM04] I. Terziev, A. Kiryakov, and D. Manov. Base upper-level ontology (bulo) guidance. SEKT deliverable 1.8.1, Ontotext Lab, Sirma AI EAD (Ltd.), 2004.

[VMP03] Yannis Velegrakis, Renee J. Miller, and Lucian Popa. Mapping adaptation under evolving schemas. In *VLDB 2003, Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany*, 2003.

[Vol04a] Raphael Volz. *Web Ontology Reasoning with Logic Databases*. PhD thesis, University of Karlsruhe, 2004.

[Vol04b] R. Volz. *Web Ontology Reasoning with Logic Databases*. PhD thesis, Institute AIFB, University of Karlsruhe, 2004.