DIP
Data, Information and Process Integration with Semantic Web Services
FP6 - 507483

Deliverable

D2.1
Report on Requirements Analysis and State of the Art
(WP2 - Ontology Management)
Version 1.00

Francisco Martin-Recuerda
Andreas Harth
Stefan Decker
Anna Zhdanova
Ying Ding
Michael Stollberg
Sinuhe Arroyo

August 31st, 2004
Executive Summary

This deliverable\(^1\) elaborates on the requirements analysis for ontology management in the DIP project. It reports on the state of the art in ontology management, points out shortcomings of existing ontology management tools, and indicates the development requirements for ontology management in the DIP project.

The technologies needed for ontology management comprise the ontology specification language used, devices for storage and retrieval of ontology specifications, versioning mechanisms for supporting evolution of ontologies, and finally devices for editing and browsing ontologies in a repository. In the course of DIP, the specific requirement is to enable handling of large-scale, geographically dispersed ontologies, allowing concurrent access and editing, and the tracking of changes in the specification of the ontologies, in order to provide versioning and evolution ontology facilities. The aim of this deliverable is to depict the requirements for achieving the outcomes we have specified here, and to outline the development directions for WP 2. Therefore, we begin by evaluating the state of the art in ontology management and show up the shortcomings of existing solutions. Taking these shortcomings into account, we then derive the development requirements for the ontology management environment of the DIP project, which will be the final outcome of WP 2.

Acknowledgements

We would like to thank Jos de Bruijn (UIB K), Carlo Wouters (VUB), York Sure (AIFB Karlsruhe), Marc Ehrig (AIFB Karlsruhe), Boris Motik (FZI Karlsruhe), Ljiljana Stojanovic (FZI Karlsruhe), Atanas Kiryakov (Sirma), John Domingue (Open University), Michel Klein (Vrije University Amsterdam), Jerome Euzenat (INRIA), Holger Knublauch (SMI), Ray Fergerson (SMI) and Natasha Noy (SMI) for useful inputs and/or discussion.

\(^1\) Disclaimer: DIP Consortium proprietary. There is no warranty for the accuracy or completeness of the information, text, graphics, links or other items contained within this material. This document represents the common view of the consortium and does not necessarily reflect the view of the individual partners.
Report on Requirements Analysis and State of the Art
(WP 2 – Ontology Management)

Document Information

<table>
<thead>
<tr>
<th>IST Project Number</th>
<th>Acronym</th>
<th>DIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP6 – 507483</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Full title Data, Information, and Process Integration with Semantic Web Services

Project URL http://dip.semanticweb.org

Document URL

EU Project officer Daniele Rizzi

Deliverable Number 2.1 Title State of the Art in Distributed Ontology Management

Work package Number 2 Title Ontology Management

Date of delivery Contractual M 6 Actual 31-Aug-04

Status Version 1.00 Final

Nature Prototype Report Dissemination

Dissemination Level Public Consortium

Authors (Partner) Francisco Martín-Recuerda, Andreas Harth, Stefan Decker, Anna Zhdanova, Ying Ding, Michael Stollberg and Sinuhe Arroyo

Responsible Author Francisco Martín-Recuerda Email francisco.martin-recuerda@deri.org

Partner UIBK Phone +43-512-507-6476

Abstract (for dissemination) This deliverable defines the requirements analysis for ontology management in the DIP project. It reports on the state of the art in ontology management, points out shortcomings of existing ontology management tools, and indicates the development requirements for ontology management in the DIP project.

Keywords Ontology Management, Ontology Storage & Retrieval, Ontology Versioning, Ontology Evolution

Version Log

<table>
<thead>
<tr>
<th>Issue Date &lt;dd-mmm-yy&gt;</th>
<th>Rev No. &lt;n.nn starting 0.10&gt;</th>
<th>Author &lt;author name&gt;</th>
<th>Change &lt;Description of the changes that were made to the preceding revision&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-Feb-04</td>
<td>0.10</td>
<td>Michael Stollberg</td>
<td>Structure Draft</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Authors</td>
<td>Status</td>
</tr>
<tr>
<td>-------------</td>
<td>---------</td>
<td>----------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>27-Apr-04</td>
<td>0.30</td>
<td>Francisco Martín-Recuerda</td>
<td>Review pre-draft</td>
</tr>
<tr>
<td>20-May-04</td>
<td>0.40</td>
<td>Francisco Martín-Recuerda and Andreas Harth</td>
<td>Review pre-draft</td>
</tr>
<tr>
<td>27-May-04</td>
<td>0.50</td>
<td>Francisco Martín-Recuerda and Andreas Harth</td>
<td>Review pre-draft</td>
</tr>
<tr>
<td>04-Jun-04</td>
<td>0.60</td>
<td>Francisco Martín-Recuerda and Andreas Harth</td>
<td>Review pre-draft</td>
</tr>
<tr>
<td>14-Jun-04</td>
<td>0.70</td>
<td>Francisco Martín-Recuerda, Andreas Harth and Anna Zhdanova</td>
<td>Pre-draft Lausanne</td>
</tr>
<tr>
<td>21-Jun-04</td>
<td>0.80</td>
<td>Francisco Martín-Recuerda, Andreas Harth and Anna Zhdanova</td>
<td>Review pre-draft</td>
</tr>
<tr>
<td>25-Jun-04</td>
<td>0.80a</td>
<td>Stefan Decker and Andreas Harth</td>
<td>Review pre-draft</td>
</tr>
<tr>
<td>02-July-04</td>
<td>0.90</td>
<td>Francisco Martín-Recuerda</td>
<td>Draft QA</td>
</tr>
<tr>
<td>11-July-04</td>
<td>0.92</td>
<td>Francisco Martín-Recuerda</td>
<td>Draft with comments of Carlo Wouters.</td>
</tr>
<tr>
<td>02-Aug-04</td>
<td>0.93</td>
<td>Francisco Martín-Recuerda, Andreas Harth and Stefan Decker</td>
<td>Draft with comments of Jos de Bruijn and Atanas Kiryakov</td>
</tr>
<tr>
<td>31-Aug-04</td>
<td>1.00</td>
<td>Francisco Martín-Recuerda,</td>
<td>Final version after proof-reading.</td>
</tr>
</tbody>
</table>
## Project Consortium Information

<table>
<thead>
<tr>
<th>Partner</th>
<th>Acronym</th>
<th>Contact</th>
</tr>
</thead>
</table>
| National University of Ireland Galway | NUIG | Prof. Dr Christoph Bussler  
Digital Enterprise Research Institute (DERI)  
National University of Ireland, Galway  
Galway, Ireland  
Email: chris.bussler@deri.org  
Tel: +353 91 512460 |
| Fundacion De La Innovacion.Bankinter | Bankinter | Jose Luis Bas  
Fundación de la Innovación. Bankinter,  
Paseo Castellana, 29  
28046 Madrid, Spain  
Email: jlbas@bankinter.es  
Tel: +34 91 6234238 |
| Berlecon Research GmbH | Berlecon | Dr Thorsten Wichmann  
Berlecon Research GmbH,  
Oranienburger Str. 32  
10117 Berlin, Germany  
Email: tw@berlecon.de  
Tel: +49 30 2852960 |
| British Telecommunications Plc. | BT | Dr John Davies  
BT Exact (Orion Floor 5 pp12)  
Adastral Park Martlesham, Ipswich IP5 3RE, United Kingdom  
Email: john.nj.davies@bt.com  
Tel: +44 1473 609583 |
| Swiss Federal Institute of Technology, Lausanne | EPFL | Prof. Karl Aberer  
Distributed Information Systems Laboratory  
École Polytechnique Fédérale de Lausanne  
Bât. PSE-A  
1015 Lausanne, Switzerland  
Email: karl.aberer@epfl.ch  
Tel: +41 21 693 4679 |
| Essex County Council | Essex | Mary Rowlett,  
Essex County Council,  
PO Box 11, County Hall, Duke Street, Chelmsford, Essex, CM1 1LX, United Kingdom.  
Email: maryr@essexcc.gov.uk  
Tel: +44 (0)1245 436524 |
| Forschungszentrum Informatik | FZI | Andreas Abecker  
Forschungszentrum Informatik  
Haid-und-Neu Strasse 10-14  
76131 Karlsruhe, Germany  
Email: abecker@fzi.de  
Tel: +49 721 9654 0 |
<table>
<thead>
<tr>
<th>Institution</th>
<th>Contact Person</th>
<th>Address</th>
<th>Email</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFI, Universität Innsbruck</td>
<td>Prof. Dieter Fensel</td>
<td>Institute of Computer Science, Technikerstrasse 13, A-6020 Innsbruck, Austria</td>
<td><a href="mailto:dieter.fensel@deri.org">dieter.fensel@deri.org</a></td>
<td>+43 512 5076485</td>
</tr>
<tr>
<td>ILOG SA</td>
<td>Christian de Sainte Marie</td>
<td>9 Rue de Verdun, 94253 Gentilly, France</td>
<td><a href="mailto:csma@ilog.fr">csma@ilog.fr</a></td>
<td>+33 1 49082981</td>
</tr>
<tr>
<td>inubit AG</td>
<td>Torsten Schmale</td>
<td>inubit AG, Lützowstraße 105-106, D-10785 Berlin, Germany</td>
<td><a href="mailto:ts@inubit.com">ts@inubit.com</a></td>
<td>+49 30726112 0</td>
</tr>
<tr>
<td>Intelligent Software Components, S.A.</td>
<td>Dr. V. Richard Benjamins, Director R&amp;D</td>
<td>Pedro de Valdivia 10, 28006 Madrid, Spain</td>
<td><a href="mailto:rbenjamins@isoco.com">rbenjamins@isoco.com</a></td>
<td>+34 913 349 797</td>
</tr>
<tr>
<td>Net Dynamics</td>
<td>Peter Smolle</td>
<td>Net Dynamics Internet Technologies GmbH &amp; Co KG, Prinz-Eugen-Strasse 68-70, A-1040 Wien, Austria</td>
<td><a href="mailto:peter.smolle@netdynamics-tech.com">peter.smolle@netdynamics-tech.com</a></td>
<td>+43 1 503982615</td>
</tr>
<tr>
<td>The Open University</td>
<td>Dr. John Domingue</td>
<td>Knowledge Media Institute, The Open University, Walton Hall, Milton Keynes, MK7 6AA, United Kingdom</td>
<td><a href="mailto:lb.domingue@open.ac.uk">lb.domingue@open.ac.uk</a></td>
<td>+44 1908 655014</td>
</tr>
<tr>
<td>SAP AG</td>
<td>Dr. Elmar Dorner</td>
<td>SAP Research, CEC Karlsruhe, SAP AG, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany</td>
<td><a href="mailto:elmar.dorner@sap.com">elmar.dorner@sap.com</a></td>
<td>+49 721 6902 31</td>
</tr>
<tr>
<td>Sirma AI Ltd.</td>
<td>Atanas Kiryakov</td>
<td>Ontotext Lab, Sirma AI EAD, Office Express IT Centre, 3rd Floor, 135 Tzarigradsko Chausse, Sofia 1784, Bulgaria</td>
<td><a href="mailto:atanas.kiryakov@sirma.bg">atanas.kiryakov@sirma.bg</a></td>
<td>+359 2 9768 303</td>
</tr>
<tr>
<td>Organization</td>
<td>Contact</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------------------------</td>
<td>------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tiscali Österreich GmbH</td>
<td>Dieter Haacker</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Tiscali Österreich GmbH.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Diefenbachgasse 35,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A-1150 Vienna,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Austria</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Email: [Dieter.Haacker(at.tiscali.com](mailto:Dieter.Haacker(at.tiscali.com)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Tel: +43 1 899 33 160</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unicorn Solution Ltd.</td>
<td>Jeff Eisenberg</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unicorn Solutions Ltd,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Malcha Technology Park 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Jerusalem 96951,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Israel</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Email: <a href="mailto:Jeff.Eisenberg@unicorn.com">Jeff.Eisenberg@unicorn.com</a></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Tel.: +972 2 6491111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vrije Universiteit Brussel</td>
<td>Carlo Wouters,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Starlab-VUB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vrije Universiteit Brussel</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pleinlaan 2, G-10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1050 Brussel, Belgium</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Email: <a href="mailto:carlo.wouters@vub.ac.be">carlo.wouters@vub.ac.be</a></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Tel.: +32 (0) 2 629 3719</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Table of Contents

1 INTRODUCTION .............................................................................................................. 1

2 ONTOLOGY MANAGEMENT ..................................................................................................... 2
2.1 Objectives for Ontology Management in DIP ............................................................ 2
   2.1.1 Large Scale Ontologies .................................................................................... 2
   2.1.2 Distributed and Multi-User Edition and Maintenance of Ontologies ............. 3
   2.1.3 Evolving Ontologies ........................................................................................ 3
2.2 Relevant Aspects of Ontology Management .............................................................. 3
   2.2.1 Ontology Language ......................................................................................... 3
   2.2.2 Ontology Storage and Retrieval ...................................................................... 4
   2.2.3 Versioning ....................................................................................................... 4
   2.2.4 Searching, Browsing, Editing, Maintenance ................................................... 5

3 STATE OF THE ART ANALYSIS ........................................................................................... 6
3.1 Ontology Language .................................................................................................... 7
   3.1.1 Overview of Representation Formalisms ........................................................ 7
      3.1.1.1 Higher-Order Logics ................................................................................ 7
      3.1.1.2 Full First-Order Logic-based Approaches ................................................ 8
      3.1.1.3 Description Logic ..................................................................................... 8
      3.1.1.4 Logic Programming and Deductive Databases ........................................ 9
   3.1.2 Evaluation Criteria ........................................................................................ 10
      3.1.2.1 Language and Technology Layering ...................................................... 10
      3.1.2.2 Built-In Primitives .................................................................................. 12
      3.1.2.3 Axiomatic vs. Integrity Constraint Semantics........................................ 13
      3.1.2.4 Pragmatics .............................................................................................. 13
      3.1.2.5 Standards/Community Support .............................................................. 13
      3.1.2.6 Summary of Evaluation Criteria............................................................. 13
   3.1.3 Evaluation of Existing Ontology Languages ................................................. 14
      3.1.3.1 KIF and the Frame-Ontology ................................................................. 14
      3.1.3.2 Topic Maps............................................................................................. 15
      3.1.3.3 Unified Modelling Language .................................................................. 16
      3.1.3.4 RDF Schema (RDFS) ............................................................................. 17
      3.1.3.5 Web Ontology Language (OWL) ........................................................... 18
      3.1.3.6 F-Logic ................................................................................................... 18

3.2 Ontology Storage and Retrieval ............................................................................... 20
   3.2.1 Evaluation Scheme ........................................................................................ 21
      3.2.1.1 Storage Schema ...................................................................................... 21
      3.2.1.2 Query Language ..................................................................................... 22
      3.2.1.3 Access Interfaces .................................................................................... 22
      3.2.1.4 Additional Features ................................................................................ 23
   3.2.2 Evaluation of Existing Ontology Storage & Retrieval Technologies ........... 24
      3.2.2.1 TAP/GetData (Stanford)......................................................................... 24
3.2.2.2 Seamark (Siderean) ................................................................. 25
3.2.2.3 RDFPeers (USC ISI) ................................................................. 26
3.2.2.4 3Store (U Southampton) .......................................................... 27
3.2.2.5 Redland Framework (U Bristol) .................................................. 29
3.2.2.6 Sesame/OMM (VU/Aduna/Ontotext) .............................................. 30
3.2.2.7 KAON (U Karlsruhe) ................................................................. 32
3.2.2.8 Jena (HP Bristol) ................................................................. 33
3.2.2.9 POSTGRES ......................................................................... 35
3.2.2.10 Subversion ............................................................................. 37

3.3 Change Management ........................................................................... 37

3.3.1 Overview ....................................................................................... 38
3.3.1.1 Merging, Mapping and Aligning Ontologies ................................. 38
3.3.1.2 Versioning and Evolution ............................................................ 45
3.3.1.3 Formalisms for Representing Changes ........................................... 47
3.3.1.4 Ontology of Change Operations .................................................. 47
3.3.1.5 Ontology Evolution Tasks .............................................................. 48
3.3.1.6 Single Ontology Evolution Management ........................................ 49
3.3.1.7 Evolution Strategy ....................................................................... 50
3.3.1.8 Distributed Ontology Evolution Management .................................. 52

3.3.2 Evaluation of Existing Tools ............................................................ 54
3.3.2.1 Ontology Middleware Module (OMM) .......................................... 54
3.3.2.2 SHOE ...................................................................................... 60
3.3.2.3 Ontoview (also known as RDFDiff) .............................................. 64
3.3.2.4 PROMPT Suite tools, an Integrated Approach ............................... 69
3.3.2.5 KAON Support for Versioning and Evolution ................................. 75

3.4 Extending Library Systems to Support Concurrent Access and Workgroup Facilities in Distributed Environment .................................................. 79

3.4.1 Overview ....................................................................................... 79
3.4.1.1 Visualization and Browsing ............................................................ 80
3.4.1.2 Concurrent Access ....................................................................... 86
3.4.1.3 Collaboration in Distributed Environment ..................................... 89
3.4.1.4 Data Integration and Neutral Ontology Representation ..................... 94
3.4.1.5 Ontology Views (Factoring) ........................................................... 95

3.4.2 Evaluation Schema ......................................................................... 97
3.4.3 Evaluation of Existing Tools ............................................................ 101
3.4.3.1 WebOnto and Tadzebao (KMI) ...................................................... 101
3.4.3.2 Ontolingua Ontology Library System (Stanford) ............................. 103
3.4.3.3 DAML Ontology Library System .................................................. 107
3.4.3.4 SHOE Ontology Library System (Maryland) ................................... 108
3.4.3.5 Protégé (Stanford) ......................................................................... 111
3.4.3.6 KAON (Karlsruhe) ...................................................................... 115

3.5 Distributed Ontologies – Practical Approaches ..................................... 118

3.5.1 Existing Practical Distributed Ontologies ........................................ 118
3.5.2 Ontology Promotion and Distribution in Practice ............................... 120
3.5.3 Simple Ontologies are not Enough .................................................. 121

4 REQUIREMENTS ANALYSIS FOR ONTOLOGY MANAGEMENT IN DIP .......... 124
4.1 Requirements Coming from Different Ontology Languages ....................... 124
4.2 Requirements for Ontology Storage and Retrieval......................................... 125
4.3 Change Management....................................................................................... 126
4.4 Requirements Ontology Library – Searching, Editing and Browsing.............. 127
5 CONCLUSION ........................................................................................................ 130
REFERENCES ........................................................................................................... 133
INTRODUCTION

One of the concerns of Ontology Management is how ontologies are stored and retrieved in a system. Ontology Management is thus concerned with techniques to make ontology data available as the knowledge base of a system, in a scalable, accurate, and secure manner. This includes the following as the most important aspects:

- **Ontology Storage and Retrieval:** Devices and mechanisms for storing ontology data securely and in a scalable manner are essential for a functional ontology management system. For usage of ontology data, they have to be retrievable with their original ontological annotations. Further, it is important that the storage and retrieval facilities support the ontological power of the ontology specification language applied.

- **Ontology Versioning:** As the world and thus ontologies evolve over time, an ontology versioning mechanism is needed to support evolution of ontologies.

- **Ontology Libraries:** Their function is mainly to support management of numerous ontologies. This mainly is concerned with tool support for editing, browsing, searching, and maintaining ontologies.

Ontology Management technologies are an essential part for ontology-based systems as they, in fact, enable the usability of ontology data as the grounding data model of the system. We may have missed some aspects in the listing above, especially in relation to ontology language and techniques for ontology integration. Of course, these are very important for the usability of ontologies, but they are not primary aspects of Ontology Management. While the expressiveness of an ontology language heavily determines the knowledge representation capabilities and thus the support for advanced information processing on basis of ontologies, for Ontology Management it is only important that the language is compatible with the storage, retrieval and versioning mechanism used. Finally, ontology integration techniques are necessary to allow interoperability of potentially heterogeneous ontologies.

The overall aim of DIP WP 2 is to develop the Ontology Management Environment of the DIP project, as an integrated platform that fulfills all tasks of ontology management that are required. The problems to be faced here are those aspects mentioned above with regard to the type of ontologies that will be dealt with in DIP. Ontologies will be used as the unique data model throughout the whole DIP system, representing domain knowledge as well as being used for describing Semantic Web Services. There will be numerous ontologies, which might be large, possibly heterogeneous, and they will evolve over time by possibly distributed editing.

At this point of time, there is no infrastructure existing which makes it possible to deal with large-scale ontologies and which integrates functionalities for secure, reliable and distributed management of ontologies. The aim of this deliverable is to examine the state of the art in ontology management techniques, concentrating on the aspects examined above, and paying special attention to conceptual solutions of existing approaches and tools, and on this basis to derive the development requirements for the DIP Ontology Management Environment. The general ordinance for this is to examine existing technologies, evaluate their strengths and weaknesses, and out of this determine the development requirements and directions for the DIP Ontology Management Environment. The basic idea is to build upon existing solutions which our analysis...
identifies as suitable for the problems and requirements arising for Ontology Management in DIP.

The deliverable is structured as follows: Section 2 defines the relevant issues in Ontology Management for DIP; Section 3 provides an exhaustive analysis of the State of the Art in Ontology Management with special attention to the requirements arising in DIP; upon this, Section 4 derives the requirements and development directions for the DIP Ontology Management Environment; Section 5 concludes the deliverable.

2 ONTOLOGY MANAGEMENT

This section defines the understanding of Ontology Management and related issues to be addressed in the workpackage. We explain why we consider an aspect to be relevant, what exactly is relevant for the workpackage for each aspect, and what the objective is for Ontology Management in DIP.

It is to remark that the integrated DIP Ontology Management Environment has to comprise more than the aspects mentioned in the following. It also has to provide access to the functionalities for reasoning on ontology data, for ontology integration, etc. which have to be evocable for applications built upon the DIP system. In DIP, the activities for ontology data handling are separated between WP 1 and WP 2: in our understanding, WP 1 is concerned with technologies for reasoning on ontology data and ontology integration (ontology mapping and aligning) – i.e. those features which are concerned with operating on ontology data during runtime of an application. WP 2 is concerned with the “repository functionality” of the DIP Ontology Management Environment, i.e. those features comparable to Database Management Systems in the world of Database-based systems. Nevertheless, there is a need for strong collaboration between these WPs in order to result in an assortative ontology infrastructure for DIP.

Consistence and correctness of ontology data provided by the ontology repository is extremely important for the system’s performance and reliability. Because of this, it is suggestive to tend to a “Conceptually Centralized Ontology Management Environment” for DIP. This means that, from a functional point of view, there is a single, centralized repository for all ontologies. Technically, this can and should be decentralized in order to support distributed and multiple edition and maintenance of data.

2.1 Objectives for Ontology Management in DIP

The overall aim of DIP WP 2 is to develop the Ontology Management Environment of the DIP project, as an integrated platform that fulfils all tasks of ontology management that are required. The following aspects arise for Ontology Management within DIP:

2.1.1 Large Scale Ontologies

Almost every component of the DIP architecture applies ontologies as its underlying data model. Domain ontologies for real-world applications can become very large, meaning they can be comprised of several hundreds of concepts and several hundred thousands of instances. Moreover, so-called system ontologies are needed in DIP in order to semantically describe the building blocks of the DIP architecture. These can
also become very large when a sizable amount of DIP components are instantiated as ontology instances.

In order to provide sophisticated support for ontology usage within the DIP-architecture, the Ontology Management Environment has to be able to deal with ontologies of such a large scale.

2.1.2 Distributed and Multi-User Edition and Maintenance of Ontologies

An essential facet of applications that utilize the Semantic Web and Semantic Web Services is that they are distributed not only in terms of the physical location of system components but also in terms of the users and entities that use, edit and maintain the ontologies.

Thus, the DIP Ontology Management Environment has to provide means and support for distributed and multi-user edition and maintenance of ontologies.

2.1.3 Evolving Ontologies

The real world evolves over time, meaning that the ontological structure of a domain can change over time. In order to ensure long-term usability of an ontology as a valid representation of a domain of discourse, ontologies have to be evolvable over time as well.

This requires a framework and mechanisms in the Ontology Management Environment that support evolution and versioning of ontologies.

2.2 Relevant Aspects of Ontology Management

The following defines the aspects of Ontology Management relevant for WP 2. For each aspect, we define the features to be addressed, provide rationales for the selection and specify the understanding and terminology to be applied throughout the deliverable.

2.2.1 Ontology Language

The formal specification language for ontologies is the essential aspect determining the quality of enhancements in information processing achievable when applying ontologies as the underlying data model in the system.

For Ontology Management, we are not interested in knowledge representation facilities or the support for reasoning (though these are issues of fundamental importance in other contexts). The important issues from the Ontology Management point of view are the following aspects:

1. **Versioning Support**

   In order to support evolvability of ontologies, the specification language has to support the versioning mechanism. More precisely, version information has to be present in the language – depending on the versioning mechanism, either in the ontology file or in each separate ontology item. Each ontology item has to be assignable to its ontology and its version.
2. Support for Importing and Aligning Ontologies

For handling large-scale ontologies, the concept of modularization is very important. Either large ontologies can be separated into smaller sub-ontologies or an ontology imports existing ontologies as a modelling basis. The language has to allow import and aligning of ontologies. The concrete way to utilize imported ontologies or parts of them depends on the framework for handling large-scale ontologies.

2.2.2 Ontology Storage and Retrieval

This aspect is concerned with storage and retrieval of ontology data, i.e. the basic “database functionality” of the Ontology Management Environment. This is very much related to the structure of the ontology specification language used. Mostly, a tool suite is used which maps ontology specifications to the persistent repository and holds the functionality for retrieval of ontology data from the repository (see e.g. Sesame architecture, [Broekstra et al., 2002]). The following are the general requirements for Ontology Storage & Retrieval systems:

1. Storage
   
   At first, storage of the ontology has to be secure and stable, which is a shortcoming of existing ontology repositories ([Gomez-Pérez, 2002], [Magkanaraki et al., 2002]). Second, a mapping of an ontology specification to a repository has to be information-conserving – meaning that when a Relational Database is used as the backend storage device, the mapping from the ontology (serialized in a file with respect to some language) to the repository has to be information-conserving according to the modelling primitives provided by the language.

2. Retrieval

   Retrieval is performed by an ontology query language which, similar to the storage facility, has to support the structure of the ontology specification language in order to allow the system to make use of the expressiveness of the ontologies. Some efforts to create ontology query languages are currently underway, but at the moment only for RDF [Alexaki et al., 2000]. For the purpose of DIP, the design of these query mechanisms has to be studied and a feasible mechanism for the ontology language to be used in DIP is needed.

It is obvious that the choice, or the design, of the ontology specification has a tremendous impact on the design of the Ontology Management facilities. Thus, before implementation of the building blocks of the DIP Ontology Management Environment can take place, the choice / design of the applied ontology specification language has to be defined.

2.2.3 Versioning

Ontology Library: Ontologies are, with respect to their nature, static representations of a knowledge structure at a certain point in time. As the world and knowledge change over time – or because the schema of an ontology is changed or extended, an Ontology
Management system has to support evolution of ontologies [Klein, 2001]. Therefore, a Versioning Mechanism is needed which handles the following aspects:

1. **Type of Change**
   Current techniques are able to identify simple (basic) and complex (composite) changes (see [Klein, 2004], [Stojanovic, 2004] and [Noy and Musen, 2003]). A simple (basic) change is any modification (add, remove and modify) of a specific feature of the knowledge model. On the other hand, a complex (composite) change is defined as composition of basic and complex changes. Versioning tools should be able to formally represent the different types of changes.

2. **Validity Check for Ontology Updates**
   The system should check whether a change leaves the ontology in a consistent state.

3. **Detection of Effects on Changes**
   Updates of an ontology might have effects on the resources that apply this ontology as their semantic schema. In order to detect such possible effects, the quality of changes needs to be observed and must be announced to the ontology users [Klein, 2001].

4. **Backward Consistency Support**
   Currently ontology versioning research or tools only focus on identifying or pointing out the changes; backward consistency support for different versions is not stressed. In the DIP Ontology Management Environment potential solutions will be exploited and researched.

5. **Repository Performance**
   In order to ensure performance of the repository, it is mandatory to think about how updates shall be displayed in the repository. See the OMM Versioning Mechanism, for example [Kiryakov et al., 2002].

### 2.2.4 Searching, Browsing, Editing, Maintenance

We group facilities for searching, editing and browsing ontologies as well as tool support for maintaining ontologies into one aspect, Ontology Library features [Ding and Fensel, 2001]. An Ontology Library provides tool support for system developers or third parties to utilize ontologies as well as to edit and maintain them. A general requirement for the DIP Ontology Management Environment is that the system supports distributed, multi-editor edition and maintenance of ontologies. For the different aspects covered by Ontology Library features we consider the following as important for DIP:

1. **Searching and Browsing**
   Search and Browsing facilities mainly provide support for system developers or third parties who want to utilize an existing ontology. Therefore, firstly, a suitable documentation of ontologies is required (incl. authors, version, ontology type classification, and information on the domain, the scope, and natural language descriptions of the main concepts). Secondly, these descriptions as
well as the ontologies have to be searchable in order to allow quick finding of needed ontologies or parts of these, because a sizable system may possibly contain numerous different ontologies. Thirdly, features for browsing specific ontologies are needed; Hyperbolic Visualization techniques are one possibility for displaying large-scale ontologies [Risden et al., 2000].

2. Editing
An editing facility has to allow modifying of ontologies in respect of the ontology specification language applied. The editing facility can be either an external ontology editor or an integrated editing functionality, or a combination of both. It is important to considerer that different ontology editing technologies sometimes apply a different “dialect” of the ontology language, meaning that the specification of an ontology with tool A might differ from the specification of the same ontology with tool B. Thus, it is important that the system can handle the language dialects of the tools applied, possibly by transforming the syntax of ontology specifications of different tools into a common, validated syntax.

3. Maintenance
Maintenance functionality is basically a workflow or tool support for updating ontologies. This is very much related to the Versioning Mechanism of the system; moreover, it shall allow distributed and multiple-party maintenance of ontologies.

4. Standards Support
Information interchange is one of the core functionalities aimed at in Semantic Web technologies. To support this, the DIP Ontology Management Environment has to support existing and emerging standards in ontology languages in order to allow information interchange and interoperability with other Semantic Web enabled applications.

3 State of the Art Analysis
Following the structure of Ontology Management aspects to be covered in DIP defined in Section 2, we will now investigate current approaches and existing tools for each of these aspects. Therefore, for each aspect, we first outline the rationale of our investigation scheme and then evaluate existing approaches and tools.

The aim of this state of the art analysis is to derive the requirements and possible starting points for the development of the DIP Ontology Management Environment. Thus, we do not provide a general analysis of existing tools and approaches – which has been provided elsewhere [Gomez-Pérez, 2002], but we concentrate on aspects and existing technologies that can provide a starting point for the DIP Ontology Management Environment. The Evaluation Schemes for the major aspects are based on the requirements outlined in Section 2, and for detailed investigation of existing technologies we restrict the evaluation to approaches that seem to be a reasonable starting point for DIP.
3.1 Ontology Language

As stated above, the ontology language to be developed or reused in DIP has a major impact on the Ontology Management solutions. Here, we discuss the requirements for the language with regard to the requirements for Ontology Management as well as those arising for reasoning and integration techniques (referring to WP 1). In fact, the determination of the requirements for the DIP ontology language is a joint effort between WP 1 and WP 2, looking into languages like OWL, and determining the suitability of existing efforts for DIP.

3.1.1 Overview of Representation Formalisms

A representation language for Ontologies has to fulfill at least the following requirements: it should be computationally and epistemologically effective. Computational effectiveness guarantees that there is an effective computational environment that can reason with Ontologies in this language. Epistemological effectiveness guarantees that the modeling process is efficient and effective, since the primitives of the language allow the entities of the domain to be expressed with minimal effort in a form which is easy and intuitive to understand. Different logic-based representation languages and inference engines have been described in the knowledge representation and reasoning literature. In the following sections, a brief review of existing approaches will be presented and their usability for inference with ontologies and metadata discussed. The review is not meant as an introduction – for this purpose we refer to the literature [Baader et al., 2003], [Lloyd, 1987] and [Enderton, 1992]. The formalisms are described in decreasing order of expressivity.

3.1.1.1 Higher-Order Logics

Higher-order logics have the greatest expressive power logics used for Knowledge Representation. In mathematics it is often necessary to talk about properties and relations as well as about entities, since this ability is required for certain reasoning and modeling tasks. By higher-order language, logicians usually mean a language in which variables are allowed to appear in places where normally predicate and/or function symbols appear.

However, there are two different facets of higher-order languages: a higher-order syntax and a higher-order semantics. For the same higher-order syntax different definitions of the semantics are possible. These semantics, usually, differ in the interpretation of their higher-order quantifiers. Since in the knowledge representation literature some confusion about these distinctions exists, the matter is investigated here in more depth.

- One possible interpretation of second-order quantifiers is the set of all relations, i.e., all subsets of the first-order universe (also called the power set of the universe). This interpretation is usually taken for second order logics by mathematicians. Unfortunately, no logical calculus can be complete with respect to this semantics, since the power set of a countable infinite universe is uncountable. Thus the set of relations of the integers is uncountable. For inference engines this results in unpleasant behavior: there are true statements, which are unprovable (This is a consequence of Goedel’s Incompleteness
Theorem [Kerber, 1994]), and thus these engines are not very useful in situations in which completeness is required.

- Another possible interpretation of second order quantifiers is the set of all definable properties, i.e., those which could be defined using an expression of the language (Henkin semantics for HOL) [Henkin, 1950]. This is only a subset of the power set, and not further investigated here.

- Yet another possible interpretation of second-order quantifiers is the set of all named properties. The named properties are the properties which have been assigned to predicate names by the interpretation. This is an even smaller subset of the power set. The resulting logic is equivalent to First-Order Logic in terms of semantics. The translation is also possible for higher-order logic in general (cf. [Goedel, 1965]).

[Chen et al., 1993] exploit the different semantics to provide a first-order semantics for a higher-order logic-programming language, resulting in a syntactically rich language which is not restricted by the computational problems of higher order logics.

3.1.1.2 Full First-Order Logic-based Approaches

Higher-order languages with higher-order semantics are often not acceptable as knowledge representation languages, since no complete inference procedures exist. Fortunately, often a first-order semantics is sufficient for real-world applications. and the resulting language provides the syntactic expressiveness of higher-order-logics, while sound and complete inference procedures can still be used.

Classical full first-order logic (FOL) (cf. [Ebbinghaus, 1994], [Enderton, 1992]) is one of the most well-known logics for knowledge representation tasks. Inference with FOL-axioms requires a fully-fledged automated theorem prover (cf. SETHEO [Ibens, 1998], OTTER, SPASS [Weidenbach, 1997], VAMPIRE [Reazonaov and Voronkov, 2002]). For FOL, sound and complete inference procedures exist, but FOL is still semi-decidable and inferencing is computationally not tractable for large amounts of data and axioms (see [Goubault, 1994] for an in-depth complexity analysis), which means full FOL is not a reasonable representation language for large ontologies. The complexity properties of full FOL are reflected by TPTP (Thousands of Problems for Theorem Prover, see [Sutcliffe and Suttner, 1998]), a collection of test-cases for theorem provers, used by the automated theorem proving community. The TPTP library contains many small, tricky problems, and theorem provers trying to solve these problems often fail [Sutcliffe, 2001].

3.1.1.3 Description Logic

Description Logics (DLs) allow specifying a terminological hierarchy using a restricted set of first order formulae. They usually have good computational properties (often decidable and tractable, and often with equally good average computational complexity), but the inference services are restricted to classification and subsumption. Given a set of formulae describing classes, the classifier associated with certain description logics will place them in a hierarchy, and given an instance description, the
classifier will determine the most specific classes to which the particular instance belongs. From a modeling point of view, DL corresponds to monadic/dyadic FOL, restricted to formulae with three variables (cf. [Borgi da, 1996]). This restriction suggests that modeling is syntactically bound. Available modern systems include FACT [Horrocks et al., 1999] and RACER [Haarslev and Möller, 2001].

Description Logics are strong in their intentional specification of ontologies (cf. [Brachman et al., 1991]), but do not provide solid support for rules and other forms of reasoning (such as procedural attachments), especially for a large volume of instance data.

Another possibility for KR tasks is languages based on Horn-logic, which is another fragment of FOL.

3.1.1.4 Logic Programming and Deductive Databases

Horn-logic is defined as the part of first-order logic where clauses are restricted to contain at most one positive literal. Horn-logic and Datalog (Horn-logic only with 0-ary function symbols) were studied in the area of deductive databases and logic programming, and a number of efficient evaluation strategies are available. Evaluation strategies can be distinguished by bottom-up and top-down variations. Top-down evaluation is usually used by Prolog systems (e.g., SWI-Prolog). However, top-down evaluation has certain disadvantages; it emphasizes the programming aspect, since a user of the system always needs to keep the evaluation strategy in mind. For instance, the clause ‘\( p(X, Z) \rightarrow p(X, Y), p(Y, Z) \)’ causes a Prolog system to loop.

Bottom-up evaluation strategies known from deductive databases (e.g., naive or semi-naive evaluation) do better; they always terminate when the set of formulae has a finite model. Goal directed bottom-up evaluation strategies (e.g., based on tabling or dynamic filtering) terminate when the model of the formulae necessary to answer the query is finite. The evaluation complexity of datalog is polynomial in the size of the database [Brewka and Dix, 1999]. Adding function symbols to recursive datalog results in an undecidable language [Brewka and Dix, 1999], however, there are many careful usages of function symbols known that preserve decidability, e.g., the usage of function symbols only with predicates which are not part of a recursive cycle.

Logic programming has been suggested as a means to Knowledge Representation (c.f. [Baral and Gelfond, 1994], [Brewka and Dix, 1999], [Baader et al., 1991]). The research has focused on extending logic programs with capabilities for non-monotonic reasoning (e.g., default negation, explicit negation, preferences) and disjunction [Brewka and Dix, 1999].

However, for knowledge representation tasks, it is not only the underlying logical formalism that is important, but also the epistemological primitives that help to represent e.g., a domain ontology. Results in object-oriented deductive databases (cf. F-Logic [Kifer et al., 1995]) provide representation primitives that can be used for knowledge representation tasks. F-Logic aims at the representation of objects, classes and relationships between objects.
3.1.2 Evaluation Criteria

3.1.2.1 Language and Technology Layering

In engineering, the layering of technologies has been used to generate a separation of concern and to make technologies more manageable. Examples include the TCP/IP stack or the ISO/OSI reference model in networking. A similar separation of concerns emerged on the Web— the W3C layer cake provides several layers, each of which is suitable for dealing with certain tasks. The layers in the layer cake directly relate to layers of reusable technology— e.g., the XML layer provides XML parsers, the RDF layer provides a storage and query infrastructure. In this section we debate which layering features are useful for enabling Ontology Management.

- Parts of an ontology (and data in general) on the Web have to be uniquely identifiable. Giving the same ID to different data items leads to confusion and conflicts. Well-known mechanisms for uniquely identifying resources on the Web are Uniform Resource Identifiers (URIs). An actual URI is not required to be able to retrieve a resource, but only to identify a person. The unique identification of elements of the ontology enables versioning support as well as storage and retrieval.

- There is more than one ontology language on the Web, and in general data on the Web is heterogeneous and often unstructured. Because of the diversity of communities using the Web and the different requirements of each of those communities, there will be no universally usable domain-specific knowledge and representation language. To still enable a joint infrastructure a common data model is necessary.

- The underlying data model must be simple and extensible. The data model serves as a foundation for other languages for all communities that exist on the Web. Therefore, the language needs to be simple and acceptable for a variety of user communities, and extensible in the sense that the user communities are able to adapt the language to their needs without compromising the entire approach. Simplicity is also required for the buy-in of user-communities — the more complicated the infrastructure is, the more difficult and expensive it is to learn and use. The layering aims to make technology more manageable.

- Ontologies and data on the Web are distributed. Data on the Web might be distributed on several servers so it should be possible to link to other sources and other data resources, and to create a web of machine-processable data. This closely resembles the idea of hyperlinks in HTML pages.

- Ontologies and data in general on the Web are biased. There is no such thing as universal truth. Anybody can say anything on the Web. The freedom to express any idea is already criminally exploited, for example with forged stock market news. So an important requirement for representing data is the ability to say something about foreign data (especially about data residing on other servers), — whether it is believable, supported, wrong, biased, etc. The requirement for a representation language is that the language needs facilities for expressing metadata about metadata (and thus statements about Ontologies).

- Support for Syntactic Interoperability. Syntactic interoperability means the ease with which data can be read and a representation obtained, such that it can be
exploited by applications. For example, software components like parsers or query APIs should be as reusable as possible among different ontology languages and other applications. Syntactic interoperability is high when the parsers and APIs needed to manipulate the data are readily available.

- Support for Semantic Interoperability. Semantic interoperability aims to address the difficulty of understanding foreign data. Semantic Interoperability is different from syntactic interoperability which talks about parsing the data, while semantic interoperability means to define mappings between unknown terms and known terms in the data. This requirement is one of the most important issues, since the cost of establishing semantic interoperability is usually higher than the cost of establishing syntactic interoperability, due to the need for content analysis.

On the other hand, it should be possible to encode everything in the data format. Since it is not possible to anticipate all potential uses, a data format must have enough expressive power to express any form of data. Clearly this requirement limits the level of support that is possible: the more domain specific a language is, the more it is possible to support the language. Models for semi-structured data are a step in the direction outlined by abstracting and simplifying relational and object-oriented data models. OEM (Object Exchange Model) is a popular semi-structured data model (cf. [Papakonstantinou et al., 1995], [Goldman et al., 1996], [Suciu, 1998]) and was developed for integration tasks in the Stanford TSIMMIS project [Garcia-Molina et al., 1995].

The RDF data model is almost identical to the data model of OEM. However, RDF takes into account various requirements for data representation imposed by the Web (e.g., URIs and linkage between data residing on different servers, namespaces etc.) that were not considered in prior approaches. RDF fulfills most of the above-mentioned requirements as a data model:

- Uniquely identifiable: URI provide a globally agreed mechanism to identify data items.
- Extensible: Communities are able to define their own vocabulary with RDF.
- Simplicity: RDF is based on one of the simplest mathematical constructs: graphs
- Distributed: URIs and URLs provide ways to distribute and link RDF data sets. More support for distributed data access is on its way (e.g., within P2P networks (e.g., [Nejdl et al., 2002]) or the W3C Data Access Working Group (see http://www.w3.org/2001/sw/DataAccess/).
- Syntactic Interoperability: Tools like Jena etc. provide support for syntactic interoperability.
- Semantic Interoperability and Integration support: global identifiers provide the hook to integrate data items. Languages like TRIPLE [Sintek and Decker, 2001] provide support for integrating different RDF data sets.
- Universal expressive power: any regular structure can be expressed in a graph, thus every information item can be captured by graphs.
So RDF is an example of a possible data model for Ontologies.

In addition, it must be realized that Ontologies are only a tiny faction of all data that is there to manage. If ontology management solutions require an entirely different set of management technologies from other kinds of data, then the effort of providing and maintaining an ontology management solution together with other solutions becomes too high. In other words, the ontology management solution should be only as specialized to Ontologies as is absolutely necessary. Since RDF is a universal data format, any ontology language can be expressed in RDF.

However, some ontology languages already come with an RDF serialization, thus no further work is required to make them available in an RDF-based ontology management system. Thus a layering in general and specifically with RDF will be an evaluation criterion for ontology languages.

3.1.2.2 Built-In Primitives

There are certain things that one may want to express in a knowledge representation language. Many ontology languages comprise object-oriented features and are based on the notion of frames [Minski, 1975]. In fact, the early frame-based languages may be considered as having been object-oriented before the term “object orientation” gained its popularity.

Semantic networks have nodes with directed labelled edges and can be represented as graphs. There exists a natural mapping between frame-based languages and semantic networks if you see objects as nodes and slots (or properties of objects) as directed and typed edges between nodes.

In logic, formulas are used to express facts, rules, and queries. The ontology languages that have a defined semantics are usually described in some form of logic (thus as formulas), as listed above.

The main features of ontology languages are listed below. Note that not all ontology languages make the distinction of the features mentioned. The following basic classification is taken from [Mylopoulos, 1980].

Classification (member-of/instance-of)

An instance can be typed by a class (John is instance of Person).

Aggregation (part-of)

An object can be composed of many other objects. (For example, a human being can be decomposed into arms, legs, head...).

Generalization (subset-of)

Sub/superclass relationships are of this kind. (Person is subclass of Living Thing).

Contexts (Partitions)

This is an important feature, which is not really addressed in most ontology languages. Certain facts need to be related to a context in order to make sense.
A more extensive set of primitives is listed in the OKBC specification [Chaudhri et al., 1998].

### 3.1.2.3 Axiomatic vs. Integrity Constraint Semantics

Different ontology languages comprise different semantics for their expression. E.g. a cardinality constraint saying that a happy father is a father with at least three children can be understood as an axiom stating truth about the world (thus requiring that in every model of the set of axioms a happy father needs to have at least three children— thus if they are not listed they must be invented). Another option is to interpret the cardinality constraints as an integrity constraint on a model— thus interpreting the data set as a model according to [Reiter, 1982] and stating whether a model fulfils the integrity constraint or not. The latter has application in data validation for Web Services — a Web Service must ensure that it gets the data to fulfil a certain task, thus validation of the send data according to a certain schema is necessary. Typically the different interpretation of statements goes hand-in-hand with another distinguishing factor: axiomatic semantic usually requires Open World semantics, whereas for checking integrity constraints a closed world viewpoint in the data is required— only the data present is validated with the ontology definitions present, not any arbitrary data. Description Logics are following a pure axiomatic approach, whereas some Logic Programming approaches follow a mixed approach: some primitives are interpreted as axiomatic (typically the subclass relationship), whereas others are interpreted as integrity constraints (e.g., cardinality constraints).

### 3.1.2.4 Pragmatics

Whereas the theoretical complexity of reasoning in certain ontology languages can be determined and is fixed, practical applicability is dependent on the existing reasoning engines and infrastructure available. However, this is in flux since new developments are becoming available every day and for evaluating ontology languages this point has only minor relevance. Since reasoning with ontologies is the topic of WP 1, we refer the reader to its deliverables for further insights on complexity issues.

### 3.1.2.5 Standards/Community Support

Ontologies as well as Ontology Languages are shared conceptualizations. Consequently, one criteria used in the evaluation section is how big the community support for ontology languages is. E.g., recommendation by the W3C is evaluated better than a single publication by a single researcher describing a particular language. However, using an argument similar to the one used in the pragmatics section, community support is in flux and their importance should not be over judged.

### 3.1.2.6 Summary of Evaluation Criteria

Based on the above discussion and description, the following evaluation criteria are established:

- **Language**: this evaluation criterion lists what the foundation of the ontology language is, i.e., on which logical language it is based.
- **Layering**: this evaluation criterion discusses which technology stack it is using (if any).
- **Built-in primitives**: this evaluation criterion summarizes the available primitives in a language.
- **Pragmatics**: this criterion lists the available tools and reasoning support.
- **Semantics**: the semantics criterion describes whether the ontology language uses axiomatic or integrity constraint semantics.
- **Standard/Community support**: standard and community support provide an overview about the acceptance of the ontology language in the community.

### 3.1.3 Evaluation of Existing Ontology Languages

In the next section, we evaluate ontology languages according to the developed evaluation schema. The following languages have been selected:

- KIF, since KIF provided the first serious effort to standardize a Knowledge Representation language.
- Topic Maps, since Topic Maps constitute a serious competitor of OWL on the Web, are standardized by ISO and have their own user community.
- UML, since the Unified Modelling Language is used by a wide user community for software development and has been standardized by the OMG.
- RDF-Schema, since RDF-Schema was the first W3C recommendation of a schema language for RDF, thus it is a predecessor of OWL and is still in widespread use.
- OWL, since OWL is the current W3C recommendation for representing Ontologies on the Web.
- F-Logic, since F-Logic is discussed in various projects (e.g., DIP itself) as a possible representation language and provides different viewpoints on Ontologies compared to OWL.

#### 3.1.3.1 KIF and the Frame-Ontology

The Knowledge Interchange Format (KIF) provides a Lisp-syntax for expressing sentences of first order predicate logic and also provides extensions for representing definitions and meta-knowledge. From a logical point of view, KIF is a highly expressive language. However, it does not provide direct representation primitives for representing Ontologies. The “Frame Ontology”, deployed in the Stanford University Knowledge Sharing Laboratory’s ontology editing tool, Ontolingua [Farquhar et al., 1997], allows users to build KIF ontologies using ontology primitives such as relations, classes (and subclasses), functions and sets [Gruber, 1993].

According the above evaluation schema, we derive the following summary:
• **Language**: KIF is actually a first-order language and the Frame-Ontology is used to rewrite expressions of the Frame-Ontology into regular KIF sentences. Therefore it inherits the computational intractability and undecidability of FOL.

• **Layering**: The Frame-Ontology is layered on top of KIF; therefore KIF infrastructure is usable also for Ontologies using the Frame-Ontology. However, KIF infrastructure is not well developed and is hard to build. In particular, there is no scalable storage and retrieval mechanism for larger amounts of KIF statement, and any other search mechanism other than doing text-search and retrieval is unsupported.

• **Built-in Primitives** defined in the Frame-Ontology are very extensive and include axioms defining the properties of slots (e.g., antisymmetric, partial-order, etc.) and properties of classes (e.g., has-instances, partition). The primitives also include a number of “management” properties like “Alias”, and “Documentation”, but no versioning primitives.

• **Pragmatics**: Tool support for the Frame-Ontology and KIF is limited—there are a number of theorem provers available (like Cyc) which are supposed to be able to deal with the Frame-Ontology using OKBC [Chaudri et al., 1998], but the number of available implementations is low and the entrance barrier to implementations is high.

• **Semantics**: KIF is centered on First-Order Logic and the meaning of the primitives is axiomatic.

• **Standards/Community Support**: KIF is a proposed ANSI standard, but was never approved. There is some support for KIF in the Knowledge Representation Community, but it has never spread far beyond.

### 3.1.3.2 Topic Maps

Topic Maps were standardized in 1999 by ISO [ISO13250, 1999]. A Topic Map is defined as a collection of Topic Map documents, which adhere to a certain SGML syntax defined in the standard document. The SGML Syntax of Topic Map documents is described in the standard, along with an informative conceptual model for memory representation of Topic Maps. Topic Maps can be used as a format for the representation of multi-dimensional subject-based indices for document collections. Topic Maps can also be used as a format for interoperable knowledge representation.

The original ISO standard specified an SGML syntax for the exchange of Topic Maps. To make Topic Maps applicable on the Web, the XML Topic Maps (XTM) standard has been defined in [XTM1.0, 2001]. XTM defines an XML syntax for Topic Maps and gives a specific, albeit slightly simplified, data model of a Topic Map. Both the SGML syntax and the XML syntax incorporate syntax shortcuts for complex data model constructs. Several representations on the object layer have been proposed for Topic Maps, the most recent one in [Garshol et al., 2003].

• **Language**: To our knowledge Topic Maps have not been formalized yet as of June 2004. Several types of formalization seem possible, among them a Full-First Order and a Horn-Logic formalization.
Layering: Topic Maps possess a data model, defined as a graph with several different kinds of nodes and arcs [Garshol et al., 2003]. Therefore there is a suitable layering but it is worth knowing that the defined data model is more complicated than the RDF graph. However, several authors (cf. [Lacher and Decker, 2001]) have proposed mappings from the Topic Maps graph model to the RDF graph model, such that RDF tools become readily available for Topic Maps as well.

Built-in Primitives: Topic Maps are not an ontology language in the classical sense— rather a language aiming to simplify the browsing of document collections using topics. Modeling primitives are available to define Topics, occurrences of Topics and Aggregations.

Pragmatics: Tool support for the Topic Maps is increasing, including a few Open Source implementations (see e.g., http://tm4j.org/). For the most part the Topic Maps development is driven by four companies: Empolis, Mondeca, Ontopia, and InfoLoom. Since Topic Maps are well defined and do not possess a complicated processing model, the entrance barrier is comparably low.

Semantics: There is no agreed semantics for Topic Maps.

Standards/Community Support: Topic Maps were developed by an open community process with open discussions and standardized by ISO. The community support is high, and since Topic Maps are comparably easy to create and deploy, spreading rate is high.

3.1.3.3 Unified Modelling Language

Unified Modelling Language\(^2\) is used in software engineering to specify programs and model specifications and use cases. UML is concerned with modelling aspects of software systems, but not with enabling computers to reason with the provided data. The language is quite expressive, with visual tools that support creation of UML via drag-and-drop user interfaces. UML is specified by the OMG (Object Management Group), an industry consortium.

Language: There is no agreed-upon formalization for UML. Several formalization attempts have been published, e.g., [Aoki et al., 2001].

Layering: UML itself is layered since it possesses a meta-model defined using MOF. However, UML and MOF are different and need different infrastructures for processing; hence the layering does nothing to make data more manageable. MOF itself is used for a variety of other purposes, e.g., in the DMTF project to define CIM (Common Information Model)\(^3\). UML is also layered on top of XML, since XMI, a serialization and exchange format for UML, is based on XML. UML has also been proposed as a visualization tool for existing ontology languages.

\(^2\) http://www.omg.org/gettingstarted/what_is_uml.htm

\(^3\) http://www.dmtf.org/education/technote_WhyCIM.pdf
• **Built-in Primitives:** UML possesses primitives for modeling classes, associations and packages and also a constraint language. Modeling primitives are not as complete as in the Frame Ontology, but seem to be sufficient for day-to-day modeling purposes.

• **Pragmatics:** Tool support for UML is plentiful, ranging from open-source tools like ARGO-UML⁴ to enterprise tools like Rational Rose⁵.

• **Semantics:** There is no agreement upon formalization for UML. Several formalization attempts have been published, e.g., [Aoki et al., 2001].

• **Standards/Community Support:** UML is a standard of the Object Management Group, which is relatively open. Development of UML is restricted to OMG members. Membership requires a fee, ranging from $550 for university affiliates to $70,000 for larger companies. While the fee for university affiliates is relatively low, for people not part of a university, the fee is usually too high to warrant participation, which means broad community support for non-university members can not be expected.

### 3.1.3.4 RDF Schema (RDFS)

For the definition of vocabularies for RDF, the RDF-Schema (RDFS) specification [Brickley and Guha, 2004] was developed. RDF-Schema is an RDF-application— it is defined in RDF itself. The defined vocabulary is very similar to the usual modelling primitives available in Frame-based languages.

• **Language:** Some semantics for RDF Schema have been developed, e.g., by Hayes [Hayes, 2004] and [Fikes and McGuinness, 2001]. [Fikes and McGuinness, 2001] is a first order semantics, but only require a limited subset of FOL. Since in [Hayes, 2004] classes and properties are also objects of the domain, [Hayes, 2004] is also a first-order semantics.

• **Layering:** RDF Schema is layered on top of RDF, therefore all RDF infrastructure is also usable for RDF Schema as well— especially storage, query and retrieval infrastructure.

• **Built-in Primitives:** RDF Schema possesses a restricted set of primitives to define classes, instances and properties. Specific versioning primitives are not present.

• **Pragmatics:** Tool support for RDF Schema is plentiful, mostly in open-source tools like Jena⁶ but some commercial infrastructure also exists (e.g. from Intellidimension.com).

• **Semantics:** Semantics as defined is axiomatic. However, an early usage of the rdfs:domain and rdfs:range primitives was not axiomatic but based on integrity constraints semantics, e.g. in early versions of the W3C RDF Validator.

---

[4](http://argouml.tigris.org/)
[5](http://www-306.ibm.com/software/rational/)
[6](http://www.hpl.hp.com/semweb/jena.htm)
Standards/Community Support: RDF Schema (like RDF) was derived in a community process by a W3C Working Group and is a W3C recommendation.

3.1.3.5 Web Ontology Language (OWL)
The OWL Web Ontology Language [Dean and Schreiber, 2004] is designed for use by applications that need to process the content of information instead of just presenting information to humans. OWL is currently being adopted by the Web community, for example in describing people in FOAF, which is based on an ontology expressed in OWL.

OWL facilitates greater machine interpretability of Web content than that supported by XML, RDF, and RDF Schema (RDF-S) by providing additional vocabulary along with a formal semantics. For deploying OWL on the web, a quite verbose RDF encoding is used. There exist more concise abstract (non RDF-based) syntaxes that are easier to read by humans but don't fit into the layered framework proposed for the Semantic Web since it doesn’t use RDF.

OWL has three increasingly-expressive sublanguages: OWL Lite, OWL DL, and OWL Full. These sublanguages can be used to create ontologies that are somewhat limited in their expressivity but have nice computational properties.

- **Language**: OWL is a language based on Description Logic. However, Disjunctive Datalog seems to be capable of capturing the semantics of Description Logics without nominals [Hustadt et al., 2004].
- **Layering**: OWL is layered on top of RDF. Therefore all RDF infrastructure is also usable for OWL as well, especially storage, query and retrieval infrastructure.
- **Built-in Primitives**: OWL possesses a number of primitives for the construction of classes and properties. OWL also features a number of primitives to deal with different versions of Ontologies, e.g., Ontology (Define an Ontology Object), imports (Import an Ontology) and versionInfo (defines a version string).
- **Pragmatics**: Jena2 provides support for OWL to some extent, but it does not support the complete semantics specification. Some DL reasoners (e.g., Cerebra (see http://www.networkinference.com/Products/Cerebra_Server.html) and Racer (http://www.sts.tu-harburg.de/~r.f.moeller/racer/)) provide support for reasoning with OWL, but so far it is unclear to what extent.
- **Semantics**: Semantics as defined is axiomatic, since OWL was derived from description logics.
- **Standard-Community Support**: OWL was derived in a community process by a W3C Working Group and is a W3C recommendation.

3.1.3.6 F-Logic
F-Logic [Kifer et al., 1995] is an extension of First-Order Logics by primitives to describe classes, objects and methods. The extensions modify the syntax as well as the semantics, but most extensions can be mapped back to conventional first-order
semantics. Based on the FOL semantics also, a semantics for a horn logic fragment of F-logic based on minimal models is defined. The language provides modelling primitives and integrates them into a logical framework. Furthermore, in contrast to most Description Logics, expressing the ontology in Frame-Logic allows queries that directly use parts of the ontology as first class citizens, i.e. variables can range over classes, instances, and attributes, and concept and attribute names can be provided as answers to queries via variable substitutions. Existing systems for the horn fragment (e.g., FLORID [Frohn et al., 1997] and FLORA and FLORA2 [Yang and Kifer, 2000]) and SiLRI [Decker et al., 1998] are tailored towards object-oriented logic programming, incorporating metaphors from PROLOG.

- **Language:** F-Logic itself is a full first-order language. However, all existing implementations all implement a subset with a slightly different semantics based on the Logic Programming paradigm.

- **Layering:** F-Logic is not layered by definition. It can be layered (and in fact most implementations work this way) on top of first-order predicate logic, which means existing infrastructure for FOL is usable.

- **Built-in Primitives:** F-Logic possesses a number of primitives for object-oriented programming, but its original definition does not include any versioning or modularization primitives. Recent implementations [Yang and Kifer, 2000] include a versioning and modularization mechanism.

- **Pragmatics:** F-Logic has a number of implementations, often in different variants. E.g., TRIPLE [Sintek and Decker, 2001] is inspired by F-Logic, but is tailored towards reasoning with RDF under multiple semantics. FLORA is a recent implementation based on XSB. SiLRI, the inference engine of Ontobroker, is a commercial variant implementing a subset of F-Logic.

- **Semantics:** Within the original language description [Kifer et al., 1995] the semantics is defined as axiomatic. However, in all implementations based on the Horn fragment the largest part of the semantics is defined to be integrity constraints based, and a smaller part is defined axiomatic based. E.g., the schemata (Ontologies) defined in F-Logic are integrity constraints for the dataset, whereas the transitivity of the subclass relationship is interpreted in an axiomatic way and therefore used to derive new facts.

- **Standards/Community Support:** F-Logic is not based on any community support. While F-Logic has been successful in science, real commercial support and widespread distribution has yet to emerge.
3.2 Ontology Storage and Retrieval

This section is concerned with storing and retrieving ontologies. We review a number of ontology stores, where some offer basic retrieval or browsing facilities (e.g. retrieval of whole ontologies, or navigation-style retrieval), and others provide query facilities with a range of query languages. Only one of the reviewed stores provides a query language with a syntax limited solely for querying on the ontology level. Most of the stores available today operate on a more general graph (RDF) model.

Every ontology language has a specific underlying data model, which means that there exist a lot of different possible ways for encoding and storing. It is not yet clear which ontology language will be used in DIP, and even if that were clear a better and more flexible solution would be to abstract from the ontology language and provide a language-neutral way of storing and accessing ontologies. If this solution were adopted, when there was a change to the ontology language, the facilities for storage and retrieval would not have to change. Moreover, defining a language-independent layer for storing and retrieval facilitates the integration of multiple ontology languages, since the same backend storage facilities can be used for a large number of languages.

We have shown in Section 3.1 that ontologies can be expressed in semi-structured data formats such as OEM and RDF. "Semi structured data is often explained as 'schemaless' and 'self-describing', terms that indicate that there is no separate description of the type or structure of data" [Abiteboul et al., 2000]. We have shown in Section 3.1 that ontologies can be expressed in semistructured data formats such as OEM and RDF. Semistructured data can be represented as edge-labelled graphs. Thus, for the following discussions, we assume that the ontologies are expressed and therefore also stored and retrieved in a semi-structured data format. Having an abstraction from the ontology language schema by using a semi-structured data format has advantages over implementing a special schema for every supported ontology language.

We have already argued that a layered approach is desirable in the overall architecture of an ontology management system in DIP. Therefore, we define that ontology storage and retrieval works at the structure level, meaning we look only at statements that are asserted, not statements that are entailed. The architecture of an ontology management system should not be monolithic, but distributed to allow for large-scale ontologies. Thus, in DIP, ontology storage and retrieval is only concerned with efficient storage and retrieval of ontologies on a structure level. The ability to perform queries on a semantic level would require performing reasoning to entail statements. Reasoning is covered in detail in WP1. We envision that a reasoner could use different ontology stores as data sources for reasoning operations, where the stores perform efficient retrieval operations of data that a reasoner requires. Implementing reasoning capabilities directly in ontology repositories would duplicate efforts in DIP and is therefore not desirable. Hence, since our definition of ontology storage and retrieval does not require reasoning, we describe in the following paragraphs storage and retrieval mechanisms for semi-structured data, in particular for data based on a graph data model.

Most of the software that can process ontologies, such as reasoners, operates on data that is held in main memory. Since ontology storage and retrieval has to deal with huge amounts of data, which needs to be stored on secondary storage such as hard disks, we
confine the discussion in this state-of-the-art analysis to stores that can operate on data stored on secondary storage devices such as hard drives.

In a nutshell, an ontology repository provides an index containing the stored data, a query language that exploits that index to access data, and interfaces to perform insert, update, remove, and query operations. While the described functionality is common to all repositories reviewed, some offer additional functionality such as support for reasoning, versioning, or access control. These features will be mentioned, but not further elaborated in this section.

3.2.1 Evaluation Scheme

We will use a combination of evaluation schemes from existing surveys ([Prud'hommeaux and Grosof, 2004], [Haase et al., 2004], [Beckett, 2002]) tailored to our purposes. The following section describes the evaluation scheme we chose. We will use the properties outlined below to structure the description of each ontology repository. We provide detailed descriptions of the indexing scheme used. This indexing scheme allows us to draw conclusions on the performance characteristics of a particular repository.

3.2.1.1 Storage Schema

Storing semistructured data means storing a graph model (directed labelled graph, triples). Means to quickly locate pieces of the graph are needed to achieve performance.

The scheme used to store semi-structured data greatly influences the performance and scalability of an ontology repository. Storage on secondary storage can be a plain text file, a relational database, or a specialized repository. Data can be maintained in main memory, on hard disk, or on a combination of both. Not all repositories reviewed offer to store data in both RAM and secondary storage.

Making data persistent in a file is the simplest way to store semi-structured data. However, accessing and querying the data is very inefficient since without additional information about where to find a particular data item the query processor has to scan over all documents to find the answer to a query. When data is made persistent in flat files, usually the entire repository is loaded into main memory and accessed from there.

Relational databases offer a convenient way to store a great variety of data. In the special case of storing semi-structured data, the graph model is mapped to the relational model and stored in a relational database (RDB). Thus, ontology repositories can use the functionality of RDBs that provide reliable and efficient data management. However, storing semi-structured data in an RDB comes with an overhead, since usually the generality the relational model offers is not needed for graph-based data. RDBs are optimized for storing relational data. Although graph-based data can be cast into the relational model, optimizations that work for the relational model might not work for this special case. Also, optimizations such as building special indices, which can speed up retrieval considerably, are not utilized since the RDB is seen as a black box.

Another class of ontology stores are specialized repositories that maintain their own records of the data together with optimized indices such as b-trees and hash tables.
RDBMS can exploit type and schema information to optimize storage and indexing, whereas storing semi-structured data is more difficult, since the data model is a directed labelled graph without any additional schema information. Schema information may exist, but since the data is semi-structured there is no guarantee that the data adheres to that schema.

Specialized indices (or tables in the case of relational systems) could be built on the ontology level, meaning that class-instance relations are indexed separately.

Provenance tracking— recording the origin of a particular piece of data— is an important feature for an ontology storage system used in a distributed environment. The Storage Schema can accommodate for provenance tracking either by providing means for reification or a context mechanism (storing quads). A related topic is truth maintenance. Knowing which facts are supporting other facts can be used in a forward-chaining reasoner to incrementally retract facts without materializing all entailments from scratch every time.

### 3.2.1.2 Query Language

The query language is used to specify a selection of parts of a data set in a declarative way. The time it takes to evaluate a query depends largely on the storage schema used, and whether there exist ways to quickly locate the required piece of data.

Query languages mostly use the triple syntax (subject, predicate, object), and are based on graph matching. As opposed to querying at the graph level, querying at the ontology level means that there are specialized and possibly optimized operations for retrieving ontology elements such as the class hierarchy. To perform querying on the ontology level correctly, inference should be performed, therefore we only briefly address ontology querying.

Multiple triple sub queries can be connected using a logical AND (conjunctive queries). Variables can by used to join the sub queries. Variables are bound during query execution and returned as result sets. For a language to achieve closure, (that means results of one query can be input for another query), either the query must return (subject, predicate, object) triples, or the query engine returns the smallest possible sub graph that is needed to extract the variable bindings for another query. If a query language provides closure, then results of queries can be used as inputs for other queries, therefore enabling a chaining of operations.

Queries can specify conditions on variables. Operators for range queries (“get all people aged between 25 and 30”) are often supported. Aggregation operators such as calculating the sum over bound variables is another useful operation in a query language. Returning variable bindings might be more convenient for the caller of the query, but lack the power of closure (and chaining operations over the network).

A brief description on how queries are evaluated is included, where available. Queries have to be reduced to either retrieve operations on the index or have to be compiled to their equivalent in SQL in cases where a relational database is used as back-end store.

### 3.2.1.3 Access Interfaces

The repository has to provide interfaces to allow for performing operations on the stored data such as inserts, updates, or queries, either over the network or from within a programming language. There are two sides to performing the operations: first, how to
invoke the operation, and second, how to encode information that is needed to perform an operation.

Operations can be invoked via an API in a programming language such as Java, C, or Python. There may exist bindings to languages such as Python, Perl, and Java to access the data stored from within programs, which can be a useful interface to the data store for quick integration with already existing systems. A program can then use an ontology repository in an embedded fashion.

Stores can provide functionality via network sockets, which are typically built on top of the API in some programming language. The ability to perform queries over HTTP is the feature most commonly supported, and some stores provide a SOAP interface as well. A few stores also support add and update operations over the network. In the Web context, HTTP as transport mechanism is preferred. In other contexts, remote procedure call (RPC) protocols such as RMI, DCOM, Corba, or SOAP might be more suited for remote access of an RDF repository.

When offering remote access to a repository, or when using the repository in a multi-threaded environment, concurrent access scenarios can occur and must be provided for. Locking the database for queries when updates are performed is an essential operation that needs to be accounted for if the indexing scheme does not support that feature. In industrial systems and large-scale databases, locking is not encouraged. Concurrent access and modifications can be better handled through transactions.

In cases where an API is used, bits of the ontology can be encoded as String or in another object in the programming language. Some repositories support more than one syntax to import data, such as RDF/XML, RDF/N3, or own formats. Parsers are needed to create a format suitable for storing in the index layer. Related to import formats is how the result of a query is returned.

3.2.1.4 Additional Features

Apart from the essential features most of the reviewed repositories have, some repositories feature additional functionality, which can include access control and authorization, versioning, support for reasoning, or features commonly used in RDBMS such as constraints and triggers.

Especially in corporate environments, access control to data is of major concern. Different categories of users will have different access rights. Users must be denied access to certain ontology parts, and should only be able to change and update data in accordance with their access rights. Access to semi-structured data must be restricted using fine-grained access controls. Fine-grained means that an administrator can specify access rights to pieces of an ontology.

Versioning is of major concern in this deliverable and is described in great detail later on. Most of the reviewed repositories don’t have versioning support.

We will mention reasoning capabilities in this section as well. However, as we have already mentioned, reasoning is not of major concern for ontology storage and retrieval.

Some repositories provide wrappers for data sources in other data formats, such as structured data in relational databases or scheduled crawling of RDF files. We don’t see these features as core features of an ontology storage and retrieval system, but mention the functionality nevertheless if available.
3.2.2 Evaluation of Existing Ontology Storage and Retrieval Technologies

Of concern in the following reviews is the core functionality of ontology storage and retrieval as outlined in the previous section. Our evaluation scheme aims at highlighting the main strengths and weaknesses of each repository. The line between pure storage mechanisms with query facilities and reasoning engines is sometimes blurred. If an ontology storage system has inference capabilities, such as Jena, KAON, or Sesame, we will mention that.

Most of the presented repositories use RDF as their underlying data format. Storage and query tools evaluated in this section cover a broad range of expressiveness and scalability: from retrieval of triples only (GetData) to fully-fledged reasoning and retrieval packages such as Jena2. We tried to start with the simpler storage and retrieval technologies and move up to the more powerful and hence more complex technologies.

Since relational database management systems offer storage and retrieval functionality for structured data, we include a review of the database technology according to our evaluation scheme as well. The major difference between RDBMS and current ontology storage solutions is that RDBMS aim at providing robust and reliable storage. The problem is that RDBMS require a fixed schema, which is not given in the semi-structured (and hence schemaless) ontology languages such as OEM and RDF. Looking at the mature RDBMS that are successfully deployed in industry to handle large amounts of data helps to shed light on shortcomings of current ontology repositories and to define requirements for ontology storage and retrieval in DIP. Most notably, access control, scalability, ensuring the integrity of the database via transactions, and robustness and reliability are the main advantages of RDBMS over ontology repositories.

In a similar vein, we include Subversion, a repository to store and retrieve documents, in our state-of-the-art evaluation. Subversion is the successor of CVS, and especially addresses versioning and distributed and concurrent changes of the same underlying document.

3.2.2.1 TAP/GetData (Stanford)

TAP/GetData [Guha and McCool, 2003] is targeted at Semantic Search, which is a small subset of query answering. However, we include TAP in this overview since it offers features that are not available in other systems, such as distributed storage and wrappers for data import. Also, TAP is a very large system, and provides insights on how to design and create the architecture for a system for storing large-scale ontologies.

The GetData query interface was designed as a lightweight alternative to more expressive RDF query languages, which can require a lot of computing power to process. In contrast to complete query languages such as SQL or XQuery, GetData only supports very basic operations, which can be implemented efficiently and be easily built into applications.

More information about TAP can be found at http://tap.stanford.edu/.

Storage Schema
TAPache, the Apache plug-in of TAP, compiles RDF files into a memory-mappable graph structure so that files do not have to be parsed when answering queries.

**Query Language**

GetData’s query language supports three operations:

- retrieve the object of a given subject and predicate
- perform keyword search on title properties
- retrieve all incoming and outgoing arcs of a node

GetData is optimized with a specific application and data set in mind, and therefore in the current version not suitable for general-purpose RDF querying, since GetData lacks some features that would need RDF features such as blank nodes.

An example for a GetData call that retrieves the current temperature in Paris, France would be (http://tap.stanford.edu/data/CityParis,_France, #temperature, ?value) and the result of the call could be 57 F.

GetData’s main feature is its simplicity, which naturally results in some limitations. Queries that involve joins can be simulated by retrieving all possible statements and performing the joins locally. However, transferring over the network a very large chunk of RDF that is not needed to answer the final query would be very inefficient.

**Access Interface**

GetData is intended to serve as a simple query interface to graph data that can be invoked over the network. GetData is built on top of SOAP. The query is issued against a URL, and the URL specifies the location where the data resides.

GetData is a query-only protocol, which means that there are no update operations defined.

**Additional Features**

TAP includes wrappers that can extract RDF out of natural text by specifying rules and a scraper that can extract information out of web pages. These data sources are regularly crawled and the resulting RDF is put into the knowledge base that is accessible via GetData. A registry is used to define what data sources are relevant to a certain query, which helps to distribute queries over different data sources. Caching is supported via the registry functionality, which caches results of queries.

**3.2.2.2 Seamark (Siderean)**

Seamark is a commercial product from Siderean Software that aims at providing powerful site search facilities for e-commerce and enterprise web sites. Seamark focuses on search within a corporate web site. [Siderean, 2003] argues that free-text search and rigid hierarchical metadata are not enough to guide users to browse a site and find relevant information. Siderean proposes the concept of faceted metadata to enhance browsing and site search. Users can restrict the instances they see to various facets or characteristics, and therefore focus their attention on a particular part of the whole data set that adheres to the selected facets. A facet can be that an instance belongs to a
certain class, or that an instance has a certain title. Multiple facets can be combined interactively, which facilitates navigating and browsing a large data set.

Seamark is a commercial product. The Siderean website at http://www.siderean.com/ has more information about Seamark.

**Storage Schema**

The data is stored in a relational database system, and the indices are optimized for answering XRBR queries, which is Seamark’s query language. The middleware runs on an Apache Tomcat servlet container, and persistent storage is provided via JDBC. More detailed information on how indices are built is not available.

**Query Language**

Seamark uses an XML document format called XRBR to represent queries and its corresponding sets of results. XRBR stands for “XML-based retrieval by reformulation”. Site navigation is expressed as a set of XRBR documents. XRBR is targeted at providing a query language to specify navigational patterns, and is therefore optimized for that purpose. XRBR includes sort facilities and defines result set sizes that facilitate the generation of a user interface similar to the user interface to search engines where users can browse through a number of results pages. XRBR is quite restricted in its functionality, and doesn’t offer general purpose RDF queries.

**Access Interface**

Seamark uses RDF as metadata format. Metadata can be loaded into the Seamark server using the SOAP protocol. Import of RDF can be scheduled at certain dates or intervals in case the underlying data changes.

**Additional Features**

Seamark provides facetted browsing and has defined its own query language to construct a user interface based on RDF stored in a relational backend.

Access control via login (admin must login, but we are not sure about other users).

Wrappers for structured data from relational databases exist, and can be scheduled for import into the Seamark database via a user interface in HTML.

3.2.2.3 **RDFPeers (USC ISI)**

RDFPeers [Cai and Frank, 2004] is a distributed RDF triple repository. The focus of RDFPeers is on providing distributed, fail-safe storage for RDF. The data is stored across a large number of nodes, and even if nodes fail, the data is still available, since the data has been replicated on other nodes in the network. The architecture is depicted in Figure 3.2.2.3.1. Although RDFPeers is not a fully-fledged RDF repository, we include it in our survey because it has interesting functionality in terms of distributed storage that none of the other RDF repositories have.

RDFPeers is not publicly available.
RDFPeers consists of a potentially large number of nodes, where each of them stores a fraction of the overall data. Nodes are arranged in a multi-attribute addressable network (MAAN). Since nodes can disappear without notice (and with them the data stored on that node), RDFPeers stores each triple at three places in the network.

RDFPeers utilizes a global hash function to determine what triple should be stored at what node. Using the global hash function, each node can determine where a certain triple has to be stored.

Data at a single node is stored in memory. Indices are kept on subject, predicate, and object. RDFPeers supports range queries and conjunctive queries.

RDFPeers provides a layered approach to querying. The simplest possible query consists of an atomic triple pattern, which is similar to the “find” operation of Jena (i.e. given subject and predicate, find the object). Next, RDFPeers supports range queries which limit the domain of variables (i.e. retrieve subject, predicate, and object where object equals “Tom”). Further, conjunctive queries are supported: queries that consist of multiple triple patterns connected via logical AND. The authors propose an algorithm to compile RDQL queries into the queries natively supported by RDFPeers; however that functionality is not implemented yet.

Queries in RDFPeers can be posed via a Java API. Import of RDF/XML is possible using Jena2’s RDF parser.

The focus of RDFPeers is on distributed storage of RDF and there is fail-safeness built-in by replicating each triple on different nodes. Functionality for load balancing is included in RDFPeers.

3Store [Harris and Gibbins, 2003] was designed for efficient handling of large RDF knowledge bases. 3Store is part of a larger system in the AKT project at http://www.aktors.org/akt/. The complete system includes a crawler that re-gathers the data nightly.
The project website at http://www.aktors.org/technologies/3store/ has more information about the software which is available under the GPL.

**Storage Schema**

The indexing structure of 3Store makes extensive use of hash tables. Model identifiers (contexts), resources, and literals are stored in separate tables with their hash as primary key. An overview of the database schema used can be found in Figure 3.2.4.1.

![Database Schema and ER Diagram Showing Table Relationships](image)

Since both resources and literals are stored using the same hash function, the main triple table contains a flag that indicates whether the object is a literal or a resource. The hash function produces a 64-bit hash to identify literal and resources. There can be collisions between literals and resources if a literal has the same value as a resource; therefore a flag is introduced to denote literals or resources. Also, key collisions are reported at assertion time to guard against two different literals or resources producing the same 64-bit hash. [Harris and Gibbins, 2003] evaluated the CRC-64 and the MD5 digest checksum hash functions for use in 3Store, with no clear outcome as to which one was a more suitable hashing function.

There is no separation between namespaces and local names in the index. The authors found that the overhead of storing namespaces in an additional (hash, uri) namespace table was too high, because the namespace prefix and local name components must be identified for each asserted resource. Separately storing namespace and prefix meant that the cost of streamed benefit of RDF/XML outweighed the benefit. The results in
3store therefore have to be accumulated in memory before creating the RDF/XML serialization.

**Query Language**

3Store uses RDQL as a query language. The 3Store RDQL query engine transforms an RDQL query into an SQL query over the underlying RDBMS indexing structure of the RDF data. The RDQL triple expressions are translated into relational calculus, which in turn is transformed into an SQL query that is executed against the MySQL database.

**Access Interfaces**

3Store features two interfaces for querying the repository: an RDQL interface that uses HTTP as transport layer and returns results in an XML format, and a database-style C API that queries the knowledge base directly. Also, a previous version of 3Store featured an OKBC-HTTP interface which allowed PHP scripts to pose queries against the data store.

There is no interface to add or update statements.

**Additional Features**

3Store’s authors claim to store 20+ million statements in 3Store. On commodity hardware, 3Store is able to assert between 300 and 1000 triples per second. 3Store is used in various AKT projects, together with user interfaces, import facilities and crawlers. Also, 3Store features reasoning facilities for RDFS.

### 3.2.2.5 Redland Framework (U Bristol)

[Beckett 2001] describes the design and architecture of the Redland RDF application framework. The framework is divided into different parts, Redland for storing triples, Rasqal for performing queries, and Raptor for parsing RDF.

The project website at http://www.redland.opensource.ac.uk/ has more information, including the software for download.

**Storage Schema**

Redland offers comprehensive RDF storage facility

Redland stores the triples using hashes. Three hash tables are used to store statements (Table 3.2.2.5.1)

<table>
<thead>
<tr>
<th>Hash</th>
<th>Key</th>
<th>Value</th>
<th>Optimized Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP2O</td>
<td>Subject and Predicate</td>
<td>Object</td>
<td>Get targets of (source, arc)</td>
</tr>
<tr>
<td>PO2S</td>
<td>Predicate and Object</td>
<td>Subject</td>
<td>Get sources of (arc, target)</td>
</tr>
<tr>
<td>SO2P</td>
<td>Subject and Object</td>
<td>Predicate</td>
<td>Get arcs of (source, target)</td>
</tr>
</tbody>
</table>

Table 3.2.2.5.1 Multiple Hash Storage in Redland [Becket 2001].
Optional indices that could be useful are incoming and outgoing arcs for a particular node. Application-specific hashes for queries that are used a lot can be included. For example, rdfs:type is a property often used, and access to that could be optimized by providing a hash index.

Redland uses Berkeley/Sleepycat DB for maintaining its indices.

**Query Language**

Redland offers a simple API to retrieve statements, or subject, predicate, or object of a given statement.

Rasqal, which is currently under development, offers query facilities for indices stored in Redland. Rasqal will support RDQL queries over Redland repositories and RDF files.

**Access Interfaces**

Redland offers APIs for Perl, Python, PHP, Java, Ruby, and Tcl.

Import interfaces and serializers for RDF/XML, N-Triples, and Turtle (a N3 dialect) are available through the Raptor parser toolkit.

**Additional Features**

Redland is a mature RDF storage and retrieval system. There exist installation packages for RedHat and Debian Linux systems. Built scripts for various Unix-architectures are provided.

3.2.2.6 **Sesame/OMM (VU/Aduna/Ontotext)**

[Broekstra et al., 2002] describe Sesame, which is a generic architecture for RDF storage and querying. More information and recent versions are available at http://sourceforge.net/projects/sesame/.

**Storage Schema**

The core of Sesame is the SAIL (Storage And Inference Layer) API, a set of Java interfaces that abstracts from the storage format and offers retrieval, inference and manipulation methods to Sesame's functional modules. SAIL implementations are available for various relational databases (MySQL, PostgreSQL, Oracle, and SQL Server) and for in-memory storage.

SAIL implements an abstract access method to perform queries to remote databases (or the persistent repositories). Sesame uses a schema that encodes parts of the RDF schema ontology. There are classes that define rdfs:subClassOf relations, rdfs:domain, rdfs:range etc.

Depending on the database, different schemas are used. The PostgreSQL backend utilizes a more complicated schema to account for PostgreSQL’s richer object-relational functionality (Figure 3.2.2.6.1), while the MySQL schema is simpler and based on the relational model.
Figure 3.2.2.6.1 Impression of the Object-relational Schema Used With PostgreSQL [Broekstra et al., 2001]

There exists also a table where all statements are stored, plus a flag that determines whether the statement was asserted or derived. The resource table is split into namespace and local name; literals are stored using a language field that denotes what language a literal is in.

[Kiryakov et al., 2002] describe an extension to Sesame that allows versioning. Versions in the repository are denoted with an update counter, which is incremented each time the content in the repository is updated. The basic operations supported are add and delete statements, and updates are modelled as delete and add operations. A history keeps a list with all add and remove operations, and therefore enables Sesame to keep track of different versions or states of the repository.

**Query Language**

Sesame supports a number of query languages, where SeRQL [Broekstra et al., 2002] is the most powerful language supported. SeRQL is developed as a hybrid language that aims to combine the strongest features of existing languages. SeRQL is one of the most expressive query languages available for RDF repositories. SeRQL draws on experiences with RQL and RDQL as well as syntax specifications such as N3. SeRQL has a CONSTRUCT clause, which allows it to perform a limited form of graph transformation.

The design of SeRQL has been based on the RDF specifications with respect to support for several features. An example of this is the addition of the 'language' and 'datatype' facets to literals, for which SeRQL has functions to query them explicitly. Also, the RDF Semantics [Hayes, 2004] has been very useful in determining the precise operation of comparison operators, for example when two literals are equal, or not. However, SeRQL currently has no support for datatype-aware comparison.

A relatively new feature of RDF Semantics is data typing. Sesame currently has limited support for data typing: it supports the use of data types but has no built-in knowledge of the semantics of XSD data types; however RDFS is not limited to XSD data types.
Access Interface
Sesame provides access through HTTP, SOAP, and RMI. For each of those protocols, the appropriate client libraries in Java are provided, which implement standard interfaces definitions. This way, an application using Sesame over SOAP, can switch to RMI with a simple change of the protocol library used. Further, those are the same interfaces provided when Sesame is embedded as a Java library for direct usage in the application. This approach provides isolation of the applications from the communication technicalities.

Sesame comes with its own RDF Parser package: RIO (RDF I/O). It supports parsing of RDF/XML and N-Triples and writing of RDF/XML, N-Triples and N3. RIOs main design considerations are speed and low memory consumption. The RDF/XML parser from the RIO package uses a SAX2-parser for parsing the XML document. Any SAX2-compliant XML parser can be used in combination with the RIO RDF/XML parser.

Sesame’s export module can serialize individuals or schema information or both into RDF/XML to enable other tools to use the contents of Sesame’s knowledge base.

[Kiryakov et al., 2002] describe access control extensions for Sesame. Access control can be granted within a repository, and within the schema, classes, instances, and properties. Patterns and queries can be used to define access rights to a set of statements that either satisfy a given pattern or are the result of a query. Different rights are supported, such as read, add, remove, admin, clear, and history. A fine-grained role system can be used to assign rights to groups of users.

Additional Features
Sesame supports the RDFS semantics through a forward chaining strategy. Each time when an update of the repository is performed, the RDF entailment rules (as specified in the RDF Semantics) are applied in order to infer all statements which could be inferred. There is a further extension, which allows “custom” defined entailment rules to be added to the basic ones. This way, one can model, for instance, transitivity of a particular property with a rule saying: \(<a,p,b> \land <b,p,c> \Rightarrow <a,p,c>\). This feature is called custom inference. Custom inference for in-memory repositories is currently under development, following requirements from several projects.

3.2.2.7 KAON (U Karlsruhe)
KAON is not exactly a store for semistructured data, but a fully-fledged ontology management system. Other functionalities of KAON are reviewed in the appropriate sections; we will limit the description in this section to KAON’s storage and retrieval facilities.

Storage Schema
KAON does not operate on the graph level, but on the ontology level. Ontologies are encoded in KAON’s class hierarchy and made persistent using Enterprise Java Beans (EJB). Therefore, no special indexing schema is used, but optimizations and storage are completely left to the EJB container and finally to a relational database.
Figure 3.2.2.7.1 Generic KAON Schema as Stored in a RDBMS [Motik et al., 2004]

**Query Language**

The query language is inspired by description logics, in a sense that the query language looks like description logics but has a different semantics.

Queries are evaluated against the EJB schema in a bottom-up fashion using magic set transformations.

**Access Interface**

Access to the system is provided via EJB, and is left to the EJB container implementation. EJB uses CORBA for remote invocation.

**Additional Features**

Provides support for reasoning, versioning, and everything else.

3.2.2.8 Jena (HP Bristol)

[Wilkinson et al., 2003] define Jena as a Semantic Web programmers’ toolkit. Jena and its successor Jena2 are widely used. Jena has a very powerful Java API that allows access to all RDF primitives from within Java programs. RDF is stored either using in-memory models, or using relational databases. We describe features from both Jena1 and Jena2 to show changes and progress in the development of the software package. We cover the storage schema in detail because Jena’s indexing schema is one of the most advanced ones used today.


**Storage Schema**

Jena1 storage in relational databases utilized three tables for storing RDF: a statement table, a literals table, and a resources table (Figure 3.2.2.8.1). The statement table stored...
all asserted and reified statements. Since each reification resulted in four statements, the storage mechanism was very inefficient (“triple bloat”). The atomic find operation was to match a pattern of (subject, predicate, object), where each S, P, O is either a constant or a do not care value. Resources and literals were referenced in the statements table using IDs, which made it necessary to use joins for the find operations.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>mylib:doc1</td>
<td>dc:title</td>
<td>Jena2</td>
</tr>
<tr>
<td>mylib:doc1</td>
<td>dc:creator</td>
<td>HP Labs - Bristol</td>
</tr>
<tr>
<td>mylib:doc1</td>
<td>dc:creator</td>
<td>Hewlett-Packard</td>
</tr>
<tr>
<td>mylib:doc1</td>
<td>dc:description</td>
<td>101</td>
</tr>
<tr>
<td>201</td>
<td>dc:title</td>
<td>Jena2 Persistence</td>
</tr>
<tr>
<td>201</td>
<td>dc:publisher</td>
<td>com1.hp/HPLaboratories</td>
</tr>
</tbody>
</table>

Figure 3.2.2.8.1 Jena2 Schema (de-normalized) [Wilkinson et al., 2003]

The Berkeley DB storage in Jena1 used a different schema. Each statement was stored in a single row and each statement was indexed by subject, predicate, and object. The Jena team observed that the Berkeley DB storage mechanism was considerably faster than relational databases.

Jena2 uses a similar schema to Jena1 to store RDF data. Only relational back-ends are supported, and the Berkeley DB support has been dropped. Literals and resources are stored directly in the statement table, which enables the find operation to work without joins. Only very large literals or resources are still stored in separate tables and referenced from the statement table, so that long values are only stored once to save space. In addition, Jena2 uses multiple statements tables to be able to group together statements that are accessed regularly, enabling tuning of the database and caching of regularly-used statements.

Optimizations can be carried out in several ways. First, there are patterns that occur quite often in RDF itself, such as reified statements that are expanded into statements that have a type property of rdf:Statement and of which subject, predicate and object are stored with the rdf:Subject, rdf:Predicate, and rdf:Object properties respectively. Sequences stored using rdf:Seq is another commonly occurring pattern. Also, application-specific patterns can be taken into account when designing a database schema for a specific use case.

Jena2 introduces the property table, which offers facilities to cluster together properties that are commonly accessed together. A property table stores all instances of properties in a graph, which means that properties stored in the property table are not stored anywhere else in the index.
The Jena2 backend offers a graph interface to the higher layers of Jena, which supports add, delete, and find graph operations. Each logical graph is implemented using a list of specialized graphs, for example a logical graph can consist of specialized graphs optimized for reification, ontology triples, and the rest. Partition of the entire graph is carried out using namespaces and properties (rdf:Subject, rdf:Predicate, rdf:Object for the reification-optimized graph, and others for the ontology optimized graph). Part of the index are so-called meta-graphs storing metadata about each logical graph.

Transaction management is not carried out in Jena, but left to the database. All operations in the Jena API are atomic to ensure database consistency.

**Query Language**

The two forms of Jena querying are find queries (that match a subject/predicate/object pattern) and RDQL query which is compiled into a conjunction of find patterns which may use variables to specify joins. The use of property tables complicates the query processing considerably.

The query processor of Jena1 converts and RDQL query into a pipeline of find operations, connected with variables to specify joins. Find operations are carried out in a loop that first evaluates one pattern and then uses the variable bindings for generating the next pattern evaluation in the pipeline. For Jena2, the goal of query processing is to translate an RDQL query into a single SQL query that can be evaluated against the RDBMS while minimizing join operations since joins are very expensive to carry out.

**Access Interface**

Jena2 provides access to an RDF repository using a Java API and using the RDQL query language. One of the strengths of Jena is its very extensive API, which gives the programmer access to all RDF constructs from within Java. Import and export facilities for N3, N-Triples, and RDF/XML are provided.

Remote access to repositories is possible via Joseki, which implements the W3C NetAPI member submission that specifies how RDF stores can be accessed via a remote interface.

**Additional Features**

Jena2 features partial support for OWL reasoning, and enjoys a wide developer community. Jena2 is an industrial-grade toolkit for RDF data management and OWL reasoning.

Jena is one of the most stable repositories reviewed, and offers the most functionality. There is an active developer community, a mailing list, and documented source code available.

**3.2.2.9 POSTGRES**

Although RDBMS are not ontology storage and retrieval systems according to our definition in the introduction, we nevertheless include a description of techniques used in these quite powerful systems on the basis of POSTGRES. RDBMS are much more mature than the reviewed ontology repositories, and therefore provide some insights on how to efficiently implement a storage and retrieval system. Since RDBMS are in
production use for a couple of decades, the features available can give a hint on what functionality is needed and used in the marketplace.

Further, the database community has some experience of how to deal with large-scale data in a commercial environment. We would like to highlight two concepts from relational database systems that may be interesting in the realm of semi-structured data: views, and constraints and triggers.

We review POSTGRES based on [Stonebraker and Row, 1986]. The current version with up-to-date information is available at http://www.postgresql.org/. In current versions POSTGRES supports SQL as query language.

**Storage Schema**

POSTGRES stores all data on disk using one relation per file. Each tuple (database record) is assigned an immutable 64-bit identifier (IID), timestamps which record when the tuples become valid (tmin) and cease to be valid (tmax), transaction identifiers (BXID, EXID), version identifiers (v-IID), and a descriptor which contains an offset at which a field starts.

POSTGRES allows users to save and query historical data. By default, data in a relation is never updated or deleted, but the old tuple is assigned new timestamps that define that it has become invalid, and the new (valid) tuple is stored in the database. A similar approach on the statement level is implemented in the OMM extension of Sesame [Kiryakov et al., 2002].

For each version, differential files contain the tuples added or subtracted from the base relation. Queries relating to versions start with the differential relation, and use the v-IID to find the version that is specified in the query. There are operations to clean the index, for instance to remove old versions. The indexing scheme of POSTGRES supports B-trees and OB-trees. Two separate indices are provided, one for the data on secondary storage (such as magnetic disk), and one for the data on tertiary storage (such as optical disk). POSTGRES has support for transactions.

**Query Language**

The query language of POSTGRES includes operations to create and destroy relations, append, delete, replace, and retrieve tuples, and define views and integrity constraints. Several data types are supported, such as integer, floating point, fixed and variable length arrays, and complex objects.

**Access Interface**

POSTGRES can be run as one process per application program, or in server mode where open file descriptors and buffers can be shared.

**Additional Feature**

Alerts, triggers, and rules are supported in POSTGRES. A trigger is a piece of code that waits for an event to occur; possible events are the insertion or deletion of a certain kind of data item. Triggers could be implemented using rules. However, these advanced features are more likely to be relevant in DIP WP1 since this deliverable is only concerned with storage and retrieval.
3.2.2.10 Subversion

In the document community, repositories that can track changes have a long tradition. Notably CVS, or its successor Subversion, offer quite sophisticated mechanisms to deal with concurrent editing, and provide versioning support for documents. Although semistructured data poses different requirements on the repository, we think it is worthwhile to briefly describe the mechanisms Subversion provides for distributed edition of documents.

The strength of Subversion is in distributed edition and version support. Subversion is available for download at [http://subversion.tigris.org/](http://subversion.tigris.org/).

Storage Schema

Subversion’s index is based on Berkeley DB. A subversion repository is a sequence of directory trees. Each tree is a snapshot of how files and directories looked at some point in time.

Query Language

The query operation supported by Subversion is quite basic: retrieve a document, either in the current version or in a previous version. Different branches of a document are also supported.

Although Subversion supports metadata, the query language is very basic: you can retrieve a document given its version.

Access Interface

WebDAV [Goland et al., 1999], an extension to HTTP, is one way to access the repository. Apache2 plug-in modules are supported for this type of access. WebDAV is described in RFC 2518 that defines a set of concepts and extensions to HTTP 1.1 for enabling more sophisticated read/write operations over HTTP. The basic idea is that a WebDAV server can act like a generic file server, and WebDAV clients can mount “shares” similar to SMB or NFS. WebDAV-compliant clients can be used to operate on a Subversion repository and check files in and out or browse the repository. DeltaV is the extension that specifies a protocol for performing incremental updates.

Authentication is performed using Apache HTTPS’s standard authentication options, based on basic authentication, X.509, LDAP, NTLM, and others.

Another access method is the native Subversion protocol via the Subversion client.

Additional Features

Hook-scripts can be specified that perform a certain actions once an event has occurred. From the functionality, hook scripts are similar to triggers in relational database systems.

3.3 Change Management

This section intends to give an overview of the most relevant ideas about merging, mapping, alignment, versioning and evolution. It also introduces some tools that cover some of the functionality that is required by these tasks.
The principal sources of information that guide us for this work are the PhD thesis of Michel Klein and the publications of Klein and colleagues (see [Klein, 2004], [Klein and Noy, 2003], [Klein et al., 2002a] and [Klein, 2001]) concerning Change Management (we borrow the title for this section), the work of Natasha Noy and colleagues (see [Noy and Klein, 2004], [Noy and Klein, 2003], [Noy and Musen, 2003], [Noy and Musen, 2003a], [Noy and Musen, 2002], [Noy and Musen, 2001] and [Noy and Musen, 2000]) with the suite of tools PROMPT and the work of Ljiljana Stojanovic and colleagues (see [Maedche et al., 2003], [Maedche et al., 2003a], [Stojanovic, 2004], [Stojanovic et al., 2003] and [Stojanovic et al., 2002]) about ontology evolution.

Four central ideas are discussed in this section. The first one is presented by Michel Klein in his PhD thesis [Klein, 2004] where he challenges the position that it is possible to distinguish between ontology versioning and ontology evolution.

The second important idea emerges from the work of Natasha Noy and colleagues with PROMPT [Noy and Musen, 2003a] where they demonstrate that versioning, mapping, merging and aligning tools can share many of the heuristics and approaches for these tasks. So they defend a global strategy for solving these tasks using an integrated approach.

The contribution of Ljiljana Stojanovic and colleagues [Stojanovic et al., 2002] in the Ontology Evolution field provides the third main part of this section. They define the principal phases in the Ontology Evolution process. They particularly stress the idea of analysing the changes proposed before those are applied in order to detect potential inconsistencies, and generating the corresponding set of complementary changes that fix the situation. The last part of the Ontology Evolution process is the Propagation of the changes to the related elements of the evolved ontology like distributed instances, dependent ontologies or dependent applications.

The last aspect that we would like to comment on in this section is that there is no general agreement about the terminology that it is used in the mediation area, and there are several interpretations of how to define and to create mechanisms to deal with merging, mapping, alignment, versioning and evolution. This situation has implications for some of the current efforts that the community are undertaking in the field (like for example, DIP, SEKT and Knowledge Web).

### 3.3.1 Overview

#### 3.3.1.1 Merging, Mapping and Aligning Ontologies

In [Klein, 2001], the author provides a set of definitions for the main tasks that are described in this section:

- **Combining**: “using two or more different ontologies for a task in which their mutual relation is relevant”
- **Merging and Integrating**: “creating a new ontology from two or more existing ontologies with overlapping parts, which can be either virtual or physical”
- **Aligning**: “bring two or more ontologies into mutual agreement, making them consistent and coherent”.
- Mapping: “relating similar (according to some metric) concepts or relations from different sources to each other by an equivalence relation. A mapping results in a virtual integration”
- Articulation: “the points of linkage between two aligned ontologies”
- Translating: “changing the representation formalism of an ontology while preserving the semantics”
- Transforming: “changing the semantics of an ontology slightly to make it suitable for purposes other than the original one”
- Versioning: “a method to keep the relation between newly created ontologies, the existing ones, and the data that conforms to them consistent”

From our interpretation of these definitions, the difference between Aligning and Mapping is that Alignment is a more complex process than mapping where it is necessary to do a deeper analysis of the ontologies. In order to maintain coherence and consistency, the definition of alignment implicitly includes the possibility of modifying the ontologies involved to guarantee this mutual agreement. So we can conclude that mapping is a prior step of alignment (or in other words, mapping could be considered as part of the alignment process). Following this reasoning we can also conclude that alignment could be the prior step of a merging process (or in other words, alignment could be considered as a part of the merging process).

On the other hand, in the context of work-package 4 of the SEKT project, their members provide several informal definitions for some of the terms discussed above:

- Mapping: define transformation rules to obtain an equivalent semantic specification of two or more ontologies. The mapping tool has to deal with the fact that these ontologies could use different representation formalisms.
- Merging: generate an integrated ontology (coherent and consistent) from two or more ontologies.
- Alignment: defines equivalences/relations between elements of two or more ontologies.

We can observe that the definitions from Klein and from the members of the SEKT project are different. Mapping in SEKT is more complex and sophisticated than mapping for Klein. In fact, mapping for Klein and alignment for SEKT are alike. So it is important to keep in mind that there are several interpretations of and definitions for the tasks that are described in this section.

Work-package 2.2 (Heterogeneity) of Knowledge Web also deals with ontology alignment and the conceptual idea of ontology alignment is also very close to the ideas that we mentioned concerning SEKT: “given two ontologies which describe each a set of discrete entities (which can be classes, properties, rules, predicates, etc.) find the relationships (e.g. equivalence or subsumption) holding between these entities”

7 http://km.aifb.uni-karlsruhe.de/projects/sekt/index.html
8 http://kw.dia.fi.upm.es/semanticportal/jsp/frames.jsp
[Euzenat et al., 2004a]. Also in an earlier draft of this paper ([Euzenat et al., 2004]),
mapping is defined as follows:

"... mapping can be described as a 5-tuple:

<id, entity1, entity2, TrustDegree, <R, n>>

where:
1. id is the identifier of the mapping;
2. entity1 and entity2 are the entities between which a relation is asserted by
the mapping (could be single entities, but also sets of entities);
3. TrustDegree is a (qualitative? quantitative?) degree of trust (confidence) in
that mapping (notice, this degree does not refer to the relation R: it is rather a
measure of the trust in the fact that the mapping is appropriate ("I trust 70% the
fact that the mapping is correct/reliable/ . . . "). The trust degree can be
computed in many ways, including users’ feedback or log analysis;
4. <R, n> is the relation associated to a mapping, where R is the name of the
relation holding between entity1 and entity2, and n is a degree of confidence
associated to the relation. Since the relation, on its basis, is model-theoretic (e.g.
equivalence, logical implication, etc.), we will assume that when n is 1 (or 0)
then the model theoretic truth-value is true (or false). For example, the
equivalence relation means equality (or inequality) if the degree of confidence is
1 (or 0), but it becomes some sort of notion of similarity for any value of the
degree of confidence strictly included between 0 and 1 ...."

A discussion between Francisco Martín-Recuerda (DERI Innsbruck), Marc Ehrig (AIFB
Karlsruhe) and Jerome Euzenat (INRIA) during the Knowledge Web Work-Package 2.2
meeting in Crete (15-05-2004) originated a debate in the KW 2.2 mailing-list that it is
summarized in the next classification:

- Ontology adaptation:
  - Bouquet: "changing an entity in one ontology to establish a (stronger)
    relation with an entity of another ontology"

- Ontology alignment:
  - Bouquet: "A bi-directional projection of two ontologies onto each other
    (I would use the static interpretation of "alignment", in the sense
    proposed by Jerome in analogy with DNA)"
  - Ehrig and Martín-Recuerda: "Define equivalences/relations between
    elements of two or more ontologies"

---

9 Summary of the contributions of Paolo Bouquet (University of Trento), Marc Ehrig (AIFB Karlsruhe),
Jerome Euzenat (INRIA), Francisco Martín-Recuerda (DERI Innsbruck) and Max Völkel (AIFB
Karlsruhe) in the KW 2.2 mailing list.
Euzenat: “alignment as a second step after mapping looking for deeper analysis of the ontologies to maintain coherence and consistency (noting that it can also imply the modifications of the ontologies to guarantee this mutual agreement). Think more about DNA-alignment.”

Euzenat interprets Spaccapietra: “He does not like it because he considers that alignment refers to some alignment against some norm (think about the non-aligned countries)”

[Klein, 2001]: “bring two or more ontologies into mutual agreement, making them consistent and coherent”.

[Kalfoglou and Schorlemmer, 2003]: “The task of establishing a collection of binary relations between two ontologies.”

Völkel: “more about bringing two things into "the same line””

- Ontology articulation:
  - Euzenat interprets [Kalfoglou and Schorlemmer, 2003]: Ω + pair of morphisms.
  - [Kalfoglou and Schorlemmer, 2003]: “an ontology that defines the binary relations between the vocabulary of two ontologies (O1 and O2), and the mappings between this articulated ontology and each of the terms of each of the two ontologies (O1 and O2).”
  - [Klein, 2001]: “the points of linkage between two aligned ontologies”
  - [Mitra et al., 2000]: “An articulation ontology denotes the semantic intersection of two source ontologies. The intersection is an operation in the so-called ontology algebra [Wiederhold, 1994]”.

- Ontology coordination:
  - Bouquet: “(instead of ‘ontology integration’, because - as Jerome pointed out - integration is already used in the db community): broadest term that applies whenever knowledge from two or more ontologies must be used at the same time in a meaningful way (e.g. to achieve a single goal).”

- Ontology evolution:
  - [Klein 2004]: “We will combine ontology evolution and versioning into a single concept defined as the ability to manage ontology changes and

---

10 One important relation which can be called approximation expresses that one ontology (a) is a representation of at least the same modelled domains as another (a/O, O’). In logic, this relation corresponds to entailment [Euzenat et al., 2004a].
their effects by creating and maintaining different variants of the ontology.”

- Stojanovic et al., 2002]: “the timely adaptation of an ontology to changed business requirements, to trends in ontology instances and patterns of usage of the ontology-based application, as well as the consistent management/propagation of these changes to the dependent artifacts (including other parts of the same ontology, in the ontology instances, dependent ontologies and application programs)”

- Ontology integration:
  - Euzenat: “This is certainly like merging but more often used for data and/or information than ontology. Moreover, merging is related to getting one resulting merged ontology while integration can be "lazy" (i.e. there is no such merged object)”.
  - Klein, 2001]: “creating a new ontology from two or more existing ontologies with overlapping parts, which can be either virtual or physical” (the same as merging).
  - Völkel: “putting one artefact more-or-less into another thereby creating a working thing. Really, it has the distinction of the artefact to be integrated and the one to be integrated with”.
  - Völkel interprets Klein: “find correspondences + assert them + check consistency”.

- Ontology mapping:
  - Bouquet: “formal statement of an exact relation between entities of different ontologies (basically, the notion of mapping we have in the current version of the document, where exact means semantically defined, and therefore covers fuzzy mappings as well)”.
  - Ehrig and Martin-Recuerda: “define transformation rules to obtain an equivalent semantic specification of two or more ontologies. The mapping tool has to deal with the fact that these ontologies could use different representation formalisms”.
  - Euzenat: “A mapping is a synonym for a function in mathematics. Providing ONE value, at most, for elements in the source set. So, I would strongly oppose using the mapping word for any stuff that allows several of them. This would be prone to misunderstandings. Unfortunately, this is the case for many alignment/matching algorithms...”
  - [Kalfoglou and Schorlemmer, 2003]: “a morphism assigning the symbols in one ontology to another”.

11 “Structured-preserving mappings between mathematical structures are called morphisms” [Kalfoglou and Schorlemmer, 2003]
o [Klein, 2001]: “relating similar (according to some metric) concepts or relations from different sources to each other by an equivalence relation. A mapping results in a virtual integration”

o Völkel: “mathematical broad term like "function"”

• Ontology matching:

  o Bouquet: “assertion of shallow similarities between entities belonging to different ontologies (this would be an explicit reminder to the notion of pattern matching, mentioned by Jerome Euzenat: shallow means not grounded on explicit and exact semantic information, e.g. string matching”).

  o Euzenat: “I do not like it very much because it is reminiscent of pattern matching where you have 1st order patterns and 0th order data to be matched. However, this might be a weak reason”.

  o Völkel: “a little more about finding out what fits together than actually doing something. Both parts are subject to change, no clear distinction between the two ontologies (no master and slave)”.

• Ontology merging:

  o Bouquet: “creating a new, third ontology out of two other ones that carries knowledge from both sources”.

  o Ehrig and Martín-Recuerda’: “generate an integrated ontology (coherent and consistent) from two or more ontologies”.

  o [Kalfoglou and Schorlemmer, 2003]: “minimal union vocabularies S₁ and S₂ and axioms A₁ and A₂ of two ontologies O₁ and O₂ previously aligned through an articulated ontology (O₀). Following the terminology of KW2.2 (see [Euzenat et al., 2004a]): pair of morphism + µ₁₂”.

  o [Klein 2001]: “Creating a new ontology from two or more existing ontologies with overlapping parts” (the same as integration).

  o Völkel: “roughly equivalent to "fusion"”.

• Ontology modularization:

  o [Stuckenschmidt and Klein, 2003]: “A modular ontology M = {M₁, … , Mₙ} is a set of modules such that for each externally defined concept C ≡ Mᵢ : Q (ontology based query over the signature of Mᵢ), Mᵢ is also a member of M”.

  o Völkel (from WP 2.1 scalability): “one ontology is broken into parts (modules) while creating some kind of relation between these modules”.

---

12 Ontology results from a merging process of two ontologies [Euzenat et al., 2004a].
• Ontology projection:
  o Bouquet: “a (partial, directional) projection of the content of one ontology onto another one”

• Ontology reconciliation:
  o Bouquet: “a bi-directional process that harmonizes the content of two (or more) ontologies, typically requiring changes on both sides”.
  o Martín-Recuerda: “I have doubts about the interpretation of ontology alignment and ontology reconciliation of Paolo Bouquet. I think that ontology reconciliation is closer to the idea of Klein about ontology alignment, and may be ontology reconciliation and alignment should be the same thing”.

• Ontology translating:
  o Euzenat interprets Klein: “Expressing an ontology o from language L into a language L' (preserving the models...)”.
  o [Klein, 2001]: “changing the representation formalism of an ontology while preserving the semantics”

• Ontology transforming:
  o [Klein, 2001]: “changing the semantics of an ontology slightly to make it suitable for purposes other than the original one”
  o Euzenat interprets Klein: “Obtaining a new ontology o' from one ontology o by using a function t”.

• Ontology Versioning:
  o [Klein, 2001]: “a method to keep the relation between newly created ontologies, the existing ones, and the data that conforms to them consistent”.
  o [Klein, 2004]: “We will combine ontology evolution and versioning into a single concept defined as the ability to manage ontology changes and their effects by creating and maintaining different variants of the ontology”.
  o Völkel (from WP 2.3. dynamics): “deals with relations between different versions in time of more-or-less the same ontology (assumption: concepts and use-case shift only slowly over time, different versions are stored as files somewhere)”.
• Using multiple ontologies:
  o Völkel (from WP 2.2 heterogeneity, the most general use case):
    “somebody finds two (or more) ontologies on the net, at a customer or
    wherever and tries to use them both at the same time in a meaningful
    way”.

We will see later in this section a brief review of the PROMPT suite that covers some of
the issues presented in the area of ontology merging, alignment and versioning. The set
of tools was designed to reuse and share several heuristics, and this work demonstrates
that there is a close relation between all these tasks. Again, this illustrates the
terminology interpretation discrepancies in the community about these issues.

3.3.1.2 Versioning and Evolution
In [Roddick, 1995], the author defined in the context of Databases the terms schema
evolution and schema versioning, as follows: Schema evolution techniques try to
guarantee that all data can be accessible from the newest schema, whereas schema
versioning allows access to data through different versions of the schema.

[Klein, 2004], presents a discussion about the differences between Schema Versioning
(databases) and Ontology Versioning. The main differences are summarized below:

• Contrary to database schema, the ontology “schema” is also used as data in
different applications (navigation menus, query schema definitions, content
structure, etc.). An important consequence of this fact is that ontology evolution
techniques should consider not only access instance data but also changes
effects in the “schema” (conceptualization, specification and representation, see
[Klein, 2004], page 86) level.

• The ontology representation languages often include more representation
primitives (they are more expressive) than the typical database schema
representation formalism. Therefore the description and management of the
changes also increases in complexity.

• As a consequence of the higher expressivity of the ontology representation
languages compared to database schemas, ontologies usually incorporate more
semantic description. This feature simplifies the inconsistency checking
mechanism for ontology evolution because ontology itself can resolve this
problem using the reasoner mechanism associated with its representation
formalism.

• Usually database schemas are not directly reused or extended, contrary to
ontologies where there are more dependent relations between versions of the
same ontology or ontologies that share common specifications. As a result of
this dependency, the impact of a change in an ontology is higher and affects all
the related ontologies including the data.

• Ontology development is in general a more de-centralized and collaborative
process than is the case for database schemas. This situation makes it difficult to
synchronize the updates that are included in the any of the different versions.
As a consequence of the differences between database schemas and ontology “schemas”, Michel Klein points out several implications for dealing with ontology versioning and evolution. One of the most important is the impossibility of distinguishing between ontology versioning and ontology evolution, which is captured in the next paragraph of his PhD thesis ([Klein, 2004]):

“The management of changes is therefore the key issue in the support for evolving ontologies. Hence, we will combine ontology evolution and versioning into a single concept defined as the ability to manage ontology changes and their effects by creating and maintaining different variants of the ontology. This ability consists of methods to distinguish and recognize versions, specifications of relationships between versions, update and change procedures for ontologies, and access mechanisms that combine different versions of an ontology and the corresponding data.”

A second relevant consequence is the redefinition of the concept of compatibility between versions. In databases the term ‘compatibility’ relates to the ability to access all data (preservation of instance data). Because the ontology schema can be used as data we cannot define compatibility in the same terms. So the characterization of the change operations has to take this feature into consideration.

An interesting implication of the richest semantic specification in ontologies is that the number of possible change operations is in general much bigger than in database schema areas. Klein argues for the necessity to define complex change operations as a composition of simple change operations that can guarantee better results during the evolution process. An example can be the transformation of instance data with less data loss (‘move a class’ is not the same as ‘delete a class’ and ‘create a class’).

The final consequence that it is addressed in [Klein, 2004] is related to the decentralized environment where ontologies are usually defined (like the Semantic Web). The evolution of an ontology can be classified as traced or untraced. In the first case we have the list of changes that the ontology undergoes during a concrete period of time. The second case is more complex because we try to guess the sequence of changes, analyzing the differences between versions.

The vision that we described before is slightly different to the approach that Stojanovic and colleagues (see [Maedche et al., 2003], [Maedche et al., 2003a], [Stojanovic et al., 2003] and [Stojanovic et al., 2002]) follow to address the problem of Ontology Evolution. Like Klein, they promote the idea of having complex change operations. On the other hand, while Klein focussed his attention in the problem of identifying changes and providing semantic specifications for the changes between versions, Stojanovic is more concerned with defining strategies that detect and solve potential inconsistencies in the new version of an ontology (as a result of the application of one or more changes) and with providing mechanisms that propagate the changes to related elements like distributed instances, dependent ontologies and dependent applications. These strategies, called evolution strategies (see [Stojanovic et al., 2002]), can be customized by the user.
3.3.1.3 Formalisms for Representing Changes

In [Klein, 2004], the author presents several possible formalisms for representing changes between two ontologies:

- **Old version and new version of an ontology.**
  In this situation, we only have the old version of the ontology and the new version, and therefore there is no explicit change information.

- **Change Logs.**
  Store the exact sequence of actions that were performed to generate from the old ontology the new version. These mechanisms provide an unambiguous and detailed specification of the changes, and can be found in several tools like Protégé and KAON. On the other hand, change logs are less effective in a concurrent edition environment, and are more difficult to implement in distributed environments.

- **Structural Diff.**
  Define a mapping between concepts and properties of the old ontology and the new version, and also identify differences between them in terms of adding or deleting concepts/properties.

- **Conceptual Relations.**
  Specify the relation between concepts and properties of the old ontology and the equivalent concepts in the new one.

- **(Minimal) Transformation Set.**
  It is the (minimal) set of change operations that specify how the old ontology can be transformed into the new one.

3.3.1.4 Ontology of Change Operations

The Ontology of change operations is the principal element of the framework described in [Klein, 2004]. Using OWL as knowledge representation formalism (the original ontology was defined in OKBC), this ontology defines a common language for change representation where any class of the hierarchy represents a specific type of change operation. There are two main categories: basic operations and complex operations.

Basic operations modify only one specific feature of the OWL knowledge model; add, remove and modify operations are provided for each element of this OWL meta model. Examples of basic change operations can be:

- Add a class as super class of the class.
- Modify the cardinality of a specific property-restriction.
- Remove a label.

On the other hand, Complex change operations are defined as a composition of other operations or as an additional knowledge about the change. These complex change
operations are specified as extensions of the ontology of basic change operations. Examples of complex change operations can be:

- The new filler is a super class of the previous filler.
- The cardinality of a slot is restricted.
- A sub tree is moved.

The use of complex change operations can provide several significant benefits:

- The changes can be expressed at a higher level, which makes the visualization and comprehension easier.
- Using complex operations we can reduce the data loss during the transformation of instance data (‘move a class’ is not the same as ‘delete the class and create it again in another place of the hierarchy’).
- The users can get a better understanding of the effects of the operations.
- Complex operations provide better machine interpretable information that can be used to resolve issues like consistency, (backwards) compatibility, etc.

3.3.1.5 Ontology Evolution Tasks

Klein defined that an ontology evolution tool should guarantee support for the following tasks ([Klein, 2004]):

- **Data Accessibility.**
  The aim of this task is to maintain the access to instance data from different versions of the ontology. To do this, Klein identified two possible strategies. The first one is to restrict the possible changes that we can apply to the ontology in order to maintain accessibility to the data. The second aims to define translation operations conducted on the data that guarantee compatibility with new versions of the ontology.

- **Consistent reasoning.**
  The idea is to introduce control mechanisms that verify that the modifications added to a new version of the ontology do not generate inconsistencies with previous specifications that were not changed.

- **Synchronization.**
  Oliver [Oliver, 2000] defines synchronization as “the periodic process by which developers update the local vocabulary to obtain the benefits of shared-vocabulary updates, while maintaining local changes that serve local needs”. In other words, synchronization allows users to maintain local versions of shared ontology, and to include the updates of the shared-ontology in these versions that also incorporate local modifications (not contemplated in other local versions or in the shared ontology).

- **Data Transformation.**
The consecutive changes that are applied in an ontology to generate a new version of it should be also applied in the data sets adapted to this new version.

- **Management of Development.**
  
  This is the ability to support verification and authorization of the changes that are applied to an ontology.

- **Definition of Evolution Strategy**
  
  This encapsulates certain policies for evolution with respect to the user’s demands. The evolution strategy tries to automate the generation of a set of additional changes that preserve the consistency of an ontology after performing a certain type of change.

- **Editing support.**
  
  Defines possible consequences of the application of the changes for the new ontology version and shows these potential consequences to the user who suggests the changes. The evaluation of these consequences creates a new set of changes which aim to preserve consistency in the rest of the new ontology version. This new set of complementary changes can have implications for the specification of a set of evolution strategies.

- **Undo/Redo facility.**
  
  Related with the previous task, the undo/redo facility should allow users to reverse the effects of the changes that were performed in the current version of the ontology that is edited.\(^{13}\)

In [Stojanovic et al., 2002], the authors agree with Klein about the necessity to maintain the consistency of the new ontology after the changes are applied and to help the user who proposes the changes to visualize the consequences that can be provoked by their application. Nevertheless, the task of preserving consistency is taken to its extreme consequences because it is necessary to ensure consistency with the distributed instances, the dependent ontologies and the dependent applications. The two first elements are addressed in ([Stojanovic et al., 2002], [Maedche et al., 2003a] and [Maedche et al., 2002]) but not the last one, which in our opinion does not in general look like a feasible task.

### 3.3.1.6 Single Ontology Evolution Management

In [Stojanovic et al., 2002] the authors define Ontology Evolution as:

> "The timely adaptation of an ontology to changed business requirements, to trends in ontology instances and patterns of usage of the ontology-based application, as well as the consistent management/propagation of these changes

\(^{13}\) In a distributed environment several users can be working with different versions of an ontology. Reconciliation facilities would be necessary to achieve consensus on new version that is capable of incorporating most the changes that the user proposed.
To cover all the aspects that Ontology Evolution requires, Stojanovic and colleagues define a set of phases (see Figure 3.3.1.6.1) that we will describe in the paragraphs which follow.

The first step is identified as Change Representation. The changes that need to be performed are analyzed and an attempt is made to generate an equivalent set of changes that take advantage of the definitions of complex changes (composite changes); these are described in an ontology of changes. These complex (composite) changes represent a group of elementary changes applied together. As discussed before, complex changes have several advantages. For example, reducing loss of data (‘moves a concept’ is not the same as ‘delete the concept and add the concept in a new location’) or reducing the number of change operations. Also they can provide a better understanding of the consequences of the changes.

The next phase is denominated Semantics of Change, and it aims to ascertain whether the set of changes that will be performed will induce inconsistencies in the ontology. To avoid these potential inconsistencies, the system generates a set of complementary changes that solve the problem.

After the Semantics Change phase, the Change Implementation stage commences. The user is presented with a list of implications of the changes and can validate or cancel them.

The last element of the process is the Change Propagation. Related instances of the ontology, the related ontologies and the related applications are analyzed. If there are any inconsistencies with regard to the new version of the ontology, the evolution system will modify (or propose modifications) to reach a new consistent state.

### 3.3.1.7 Evolution Strategy

The previous section defined the “Semantics of Change” phase as a part of the Ontology Evolution Process. This stage aims to analyze the changes that were proposed for the
ontology and detects whether the new version of the ontology can lead it to an inconsistent state. When it detects that a change can generate an inconsistent state, the Semantics of Change phase proposes a set of complementary changes to solve the situation. There can be several sets of complementary changes that can lead the ontology to different consistent states. This part of the process can be regarded as a kind of workflow for change resolution with different possible paths that lead to several possible solutions. In this structure we can distinguish between junctions (places in the workflow with several alternatives to follow) and branches (any of the alternatives that start from a junction). In [Stojanovic et al., 2002], the junctions are called resolution points and the branches are called elementary evolution strategies.

To clarify the meaning of resolution point and elementary evolution strategy [Stojanovic et al., 2002] proposed an example of deleting a concept C embedded in a concept hierarchy (see Figure 3.3.1.7.1). The authors distinguish the following resolution points that will lead the ontology to a new consistent state:

- If there are orphaned sub concepts of C, a set of elementary evolution strategies should be performed.
- If there are properties that sub concepts of C inherit from the parents of C, a set of elementary evolution strategies should be performed.
- In the case of all properties whose domain is C, the system should provide a set of elementary evolution strategies to deal with this obstacle.
- The same problem exists for the properties whose range is C.
- If the concept C has associated instances, a set of elementary strategies should be performed.
- The instances of other concepts can have relations with instances of the concept C, so a set of elementary evolution strategies should be performed to solve this situation.

![Figure 3.3.1.7.1: Resolution Points for Deleting Concept C: a) Original ontology; b) Connection to the parent concept; c) Connection to the root concept; d) Deletion of the sub concepts [Stojanovic et al., 2002]](image)

In the case of the first resolution point, which deals with orphaned sub concepts of C, the system proposed the following set of resolution strategies (see Figure 3.3.1.7.1):
• Connect the orphaned sub concepts to the parent concept/s of C.
• Connect the orphaned sub concepts to the root concept.
• Delete the orphaned sub concepts.

3.3.1.8 Distributed Ontology Evolution Management

In [Maedche et al., 2003a] the authors debate which would be the best solution to maintain distributed ontologies in the WWW context. They conclude that replication can be the best proposal because it offers an acceptable performance using the current network infrastructure, and taking issue with Klein, they do not consider his proposal of a modular ontology architecture (see [Klein, 2004]).

The Webopedia Computer Dictionary (http://www.webopedia.com/) defines replication as:

“The process of creating and managing duplicate versions of a database. Replication not only copies a database but also synchronizes a set of replicas so that changes made to one replica are reflected in all the others. The beauty of replication is that it enables many users to work with their own local copy of a database but have the database updated as if they were working on a single, centralized database. For database applications where users are geographically widely distributed, replication is often the most efficient method of database access.”

In the context of distributed ontology evolution only the “master” ontology can be modified and the ontology management system has to provide the appropriate mechanisms to propagate the changes to the corresponding replicas. Maedche and colleagues article ([Maedche et al., 2003a]) discussed two well-known approaches to achieve this in the areas of databases:

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

They also provide convincing arguments to justify that in a distributed context the Pull-based approach is more suitable especially given the current constraints of the communications infrastructure. However, a hybrid approach can be a more flexible solution. Push-based or Pull-based will be applied depending on the particular circumstances (like for example, a necessity to update all the replicas with essential changes).

The next component of their proposal is the extension of the Ontology Evolution Process presented in previous section to adapt it to the particularities of a distributed environment. They introduced a new phase before “Change Representation” called “Change Capturing” that will identify which changes should be applied to any of the replicas (the replicas may be in different states of the evolution because it may not be possible to apply the replication process with the same frequency to all of the replicas). The set of changes that have been applied to the original ontology since the last synchronization of a replica is called deltas, and for each replica the system maintains
an instance of a special evolution log ontology (see Figure 3.3.1.8.1). This ontology models what changes, why, when, by whom and how they are performed in an ontology ([Maedche et al., 2003a]).

After the deltas are identified and extracted (extraction of deltas), the system analyzes the changes to avoid duplicate modifications (merging deltas) motivated by the existence of references to particular ontologies inside of an ontology (ontology inclusion is a useful mechanism for reusing ontologies described in [Maedche et al., 2003] and [Heflin and Hendler 2000]). All these steps are also included within the “Change Capturing” phase. Figure 3.3.1.8.2 shows the extended evolution process for distributed ontologies.

The original framework for single ontology evolution and the extension for distributed ontologies presented in this section were implemented within KAON14, an ontology management system developed at FZI and AIFB at the University of Karlsruhe.

14 http://kaon.semanticweb.org/
3.3.2 Evaluation of Existing Tools

This section includes a description and evaluation of several tools with relevant features for ontology versioning and/or evolution. The main goal is to present an overview of the current implementation state of the Change Management framework described in previous section. The list of requirements presented in Section 4.2.3 as a result of the combination of the list of evolution tasks listed in previous sections and the formalisms for representing changes proposed in [Klein, 2004] are the guidelines for the evaluation of the tools included in this section. OMM (Ontology Middleware system, [Kiryakov et al., 2002], [Kiryakov et al., 2002a]), SHOE (“Simple HTML Ontology Extensions”, ([Heflin et al., 1999], [Heflin and Hendler, 2000]), Ontoview [Klein et al., 2002a], PROMPT suite [Noy and Musen, 2003a] and Karlsruhe Ontology tool suite (KAON, http://kaon.semanticweb.org).

3.3.2.1 Ontology Middleware Module (OMM)

OMM, Ontology Middleware Module system ([Kiryakov et al., 2002], [Kiryakov et al., 2002a]), is an extension of Sesame [Broekstra et al., 2002] for knowledge management solutions. The main focus is to provide an infrastructure for development, management, maintenance, and use of large and middle-size ontologies. These are the principal features of this system:

- A scalable and reliable basic storage services (already available in Sesame).
- Knowledge Control System (KCS), an administrative software infrastructure that provides access control, changes tracking (versioning) and meta-information embedded features for knowledge bases.
- Pluggable reasoning modules suitable for several distinct domains and applications. OMM offers a specific reasoner for RDF(S) and DAML+OIL knowledge bases called BOR.
- Flexible access to the system through the support of several remote access protocols.

Figure 3.3.2.1.1 shows how the different elements of OMM are integrated in the Sesame architecture. The Storage and Inference Layer (SAIL) of Sesame is extended with new OMM functional modules represented in grey color (see more details in [Kiryakov et al., 2002], [Kiryakov et al., 2002a]).
As we mentioned before, KCS is the administrative module of OMM that provides support for versioning (tracking changes), access control and meta-information for knowledge bases. These features are interrelated to each other as depicted in Figure 3.3.2.1.2.

The Tracking Changes component of KCS follows the spirit of source control systems (such as CVS, Concurrent Versions System [Fogel and Bar, 2001]) for software development. The versioning model proposed is designed to support RDF repositories, and this decision is very much influenced by the fact that Sesame is an RDF repository. As a result of this choice, a set of consequences are outlined by the authors of OMM [Kiryakov, et al. 2002a]:

- “The RDF statement is the smallest directly manageable piece of knowledge”. Although at first sight the smallest pieces of knowledge can be resources and literals, Kiryakov and colleagues argued that there is no way to add, remove or update (the description of) a resource or a literal without also changing some statements.
• "An RDF statement may not change – it can only appear or disappear".
For Kiryakov and colleagues, changing an RDF statement is to convert the previous statement to another statement. An RDF statement is a triple and the only way to identify a triple is through its constituents.

• "The two basic types of updates in a repository are addition and removal of statements".
This is a direct consequence of the principle mentioned above. So addition and removal are the only events that necessarily have to be tracked by the system.

• "Each update turns the repository into a new state".
For Kiryakov and colleagues a “state” of the repository is defined by the set of statements that are stored, so any change in the repository turns it into a new state. A tracking system should be able to address and manage all the states of a repository.

The basic idea behind the tracking changes system of OMM is to generate for each repository an integer counter variable, update counter (UC), that increases its value each time that the repository is updated (addition and removal of RDF statements). Each separate value of UC together with the respective update operation and the statement that was affected is called update identifier (UID), and follows the following pattern:

UID: nn {add|remove} <sub, pred, obj>

A UID also has a very interesting property: it identifies also each state of the repository, as can be seen in the following example (see Figures 3.3.2.1.3, 3.3.2.1.4 and 3.3.2.1.5):

![Diagram of RDF Repository and History of Changes](image)

**Figure 3.3.2.1.3: Example of RDF Repository, and History of Changes [Kiryakov et al., 2002a]**
The information that KCS stored about tracking changes (as well as security information) is represented in RDF proprietary schema. The idea is to encode this information via a kind of special properties and classes. The hierarchy of properties together with the domain and range restrictions for each property is defined as follows [Kiryakov et al., 2002a]:

```
metalInfo
  trackingInfo  (domain=rdfs:Statement  range=)
    bornAt      (domain=rdfs:Statement  range=Update)
    diedAt      (domain=rdfs:Statement  range=Update)
  securityInfo
    lockedBy    (domain=rdfs:Statement  range=User)
```

where `trackingInfo`, `bornAt` and `diedAt` are equivalent to UID:nn, add and remove for the previous definition of UID (update identifier).
This RDF schema for representing changes in the repository is codified internally in two tables that are stored in a RDBMS (Relational Database Management System). Figure 3.3.2.1.6 shows the conceptual schema of the internal representation of the changes.

The table “Updates” keeps information relevant to each update: the time when it happened and the user who performed the change. On the other hand, the table “Triples Hist” stores for each row a UID that corresponds with an update in the repository.

We conclude this section with an analysis and comparison of versioning and evolution capabilities in OMM with respect to the requirements that were proposed for the DIP project (see Section 4.2.3, which describes the requirements for versioning and evolution):

- **Version identification.**
  OMM defines an update counter to track changes, so for any new update in the repository the counter is increased. Kiryakov and colleagues proposed that the related update identifier can be used to identify each version of the ontology in the repository. In our opinion, this simple method is not enough to identify properly the versions of an ontology, because the repository can contain several ontologies, and the tracking system only traces changes at the repository level. So it will only work well if the repository contains a single ontology. In general, this is a change-tracking identification mechanism, not an ontology versioning one.

- **Version storage.**
  The versions are not independently stored in different locations. There is only one version (the last one), and to retrieve a previous version, it is necessary to
undo the update operations until you reach the state of the repository that corresponds with the version required\textsuperscript{15}.

- **Data Accessibility**
  The aim of this task is to maintain access to instance data from different versions of the ontology, but although this is also one on the requirements of OMM (see Section 2.2 of [Kiryakov et al., 2002a]); there is no explicit description of OMM features relating to this.

- **Formal Change Representation**
  Kiryakov and colleagues justify a very simple representation of changes because the system deals with RDF repositories and the kind of operations that it is possible do with this representation mechanism are only additions and deletions of RDF statements. These operations are stored in log tables with the structures already described (see Figure 3.3.2.1.6).

- **Consistent reasoning.**
  OMM includes a reasoner called BOR that can detect potential terminological/instance inconsistencies in the ontology after the performance of some change operations.

- **Data Transformation.**
  In the context of OMM, there is no difference between instances and terminological modifications because in the end everything is RDF statements modifications. So, there is no mechanism defined in OMM to transform instances in order to be adapted to a concrete version of an ontology.

- **Security support.**
  KCS provides access control facilities for RDF repositories. Each user is registered in the system and has an associated role (or set of roles) that defines what kind of operations a particular user or class of users is able to do. Also it is possible to define security access rights for the elements of the knowledge base to a very fine-grained level (classes, properties, instances, etc).

- **Definition of Evolution strategies.**
  OMM does not provide mechanisms for encapsulateing certain policies for evolution with respect to the user’s demands.

- **Editing Support.**
  When an update in the repository is performed, OMM does not analyze the consequences of this change for the consistency of the knowledge base and does not suggest complementary changes to reach a new consistent state. Kiryakov and colleagues point out that this is the responsibility of Sesame and not of OMM, and that there is some work done in this direction.

- **Undo/Redo facility.**

\textsuperscript{15} The authors of OMM expressed their disagreement in this point: “\textit{Not true. It is straightforward to take any old state of the repository. There is a method workWithState(stateID) which transfers the repository into R/O mode and you can make any queries.}”
[Kiryakov et al., 2002a] does not explicitly describe undo/redo facility support, but the authors confirmed to us that the OMM provides this facility which is described in deliverable D39 of the OntoKnowledge project16.

OMM is an ontology middleware system developed in 2002 as an extension of Sesame [Broekstra et al., 2002] provides an infrastructure for development, management, maintenance, and use of large and middle-size ontologies. The Knowledge Control System (KCS), an administrative software infrastructure for OMM, includes a Tracking Changes module for RDF repositories. Using a counter variable, update counter (UC), that increases its value any time that the repository is updated, and a history record of update identifiers (UID), it is possible to recover previous versions of an ontology. Unfortunately, desired features like single-ontology version identification, independent version workspaces, data accessibility checking mechanisms, complex change definitions and evolution techniques are not supported. On the other hand, OMM is the only ontology management system with a comprehensive, fine-grained, access support.

3.3.2.2 SHOE

SHOE [Heflin et al., 1999] and [Heflin and Hendler, 2000], the acronym of “Simple HTML Ontology Extensions”, is an ontology-based knowledge representation language that is embedded in web pages. The main goal of SHOE was to improve the Web search mechanism, and to achieve this, SHOE was developed as an extension of HTML with the aim of including machine readable semantic knowledge in Web documents. SHOE allows representation of concepts, their taxonomies, n-ary relations, instances and deduction rules. There are several tools that are able to provide inference capabilities. One of the most interesting is Exposé ([Heflin and Hendler, 2000a]), a Web crawler for web pages with SHOE annotations that uses Parka ([Evett et al., 1994]) or XSB ([Sagonas et al., 1994]) as KR systems.

SHOE provides several mechanisms for versioning and reusing ontologies. For versioning, SHOE maintains each version of the conceptual specification of an ontology definition in a separate web page together with an identifier. The new versions can include a label that indicates that they are compatible with previous versions (backward-compatible). The case of the instances of an ontology (and its versions) is particular because they are distributed on the Web and they can provide contradictory information that SHOE is able to handle.

16 http://www.ontoknowledge.org/
On the other hand, SHOE allows users to re-use ontologies during the definition process of creating new ontologies. With the appropriate labels is it possible to refer to a specific ontology and to use concepts and relations of the referred ontology. Because in a distributed environment, ontologies can progressively diverge during their evolution, to improve the interoperability when the related ontologies started to diverge conceptually, the authors of SHOE proposed the use of its inference rules to specify mapping relations between ontologies (mapping rules). Also, they suggest three methods for achieving ontology integration:

- Mapping Ontologies.
  
  In this approach a new ontology is created. It extends several other ontologies and solves the domain differences (terminology\(^{17}\), scope\(^{18}\), encoding\(^{19}\) and context\(^{20}\)).

- Mapping Revisions.
  
  The idea is to create new versions of any of the ontologies that we want to integrate and to include in all of them the mapping to the others.

- Intersection Ontology.

---

\(^{17}\) Terminology: different names are used for the same concepts [Heflin and Hendler, 2000].

\(^{18}\) Scope: similar categories that do not match exactly [Heflin and Hendler, 2000].

\(^{19}\) Encoding: the valid values for a property can be different [Heflin and Hendler, 2000].

\(^{20}\) Context: a term in one domain has a different meaning in another [Heflin and Hendler, 2000].
The last approach is to merge the common items into a more general ontology, called intersection ontology. Subsequently, new versions of the mapping ontologies that include a reference to the intersection ontology are created.

The ability of SHOE to characterize mapping descriptions is very valuable for versioning because it makes it possible to include an explicit specification of the relations between versions.

Following the schema of the previous section, we include an analysis and comparison of versioning and evolution capabilities in SHOE with respect to the requirements that were proposed for the DIP project:

- **Version Identification**
  Any ontology and each version of an ontology has a unique identifier. The specification of SHOE\(^\text{21}\) does not include a specification of how to generate these unique identifiers. The version ID only can contain digits and dots ([Heflin, 2001], see Figure 3.3.2.2.2 with the schema specification of an ontology).

```xml
<ONTOLGY ID="id"
VERSION="version"
[BACKWARD-COMPATIBLE-WITH="bcw_1 bcw_2 ...bcw_n"]
[DESCRIPTION="text"]
[DECLARATORS="dec_1 dec_2 ...dec_m"]>
  content
</ONTOLGY>
```

Figure 3.3.2.2.2: Ontology Definition in SHOE [Heflin, 2001]

- **Version Storage**
  SHOE maintains each version of the conceptual specification of an ontology definition in a separate web page together with an identifier.

- **Data Accessibility**
  SHOE does not provide tools to guarantee the accessibility to instance data from different versions of an ontology, and although it is possible to include information in the specification of a version of an ontology showing that backward compatibility is supported, there are no integrated mechanisms to check this compatibility.

- **Formal Change Representation**

There is no formal specification mechanism to identify and describe the changes between versions of the same ontology. With the mapping rules the user has an important degree of flexibility, but the specification of the differences between versions can be very influenced by the point of view of the user. In addition, the tool does not provide automatic heuristics to help the user with the process of detecting the differences and providing the required mapping rules, and although it is possible to import ontologies that are specified in other languages, the mapping rules are very dependent on SHOE language and thus limited.

- **Consistent reasoning**
  Although SHOE is supported by several platforms for storing and reasoning with ontologies (like Parka or XSB), we could not find in the bibliography explicit references concerning control mechanisms that verify that the modifications added to a new version of an ontology do not generate inconsistencies with the rest of the specification.

- **Data Transformation**
  There is no automatic mechanism to adapt data to the modifications in the structure of a new version of an ontology (maintaining the compatibility with previous versions). Klein defined this capability as transparent translating (see [Klein, 2001]).

- **Security Support**
  In [Heflin and Hendler, 2000] the authors address the necessity to support verification and authorization of the changes that are applied to an ontology. They proposed three alternatives to tackle the problem, but none of them were implemented. Further, authorization is just the basic security functionality and nothing more comprehensive than this is specified in SHOE.

- **Definition of Evolution Strategies**
  SHOE does not provide automatic evolution strategies where the system helps users to maintain the coherence of the system after performing some changes.

- **Editing Support**
  There is no decision support component that shows the user the possible implications of a change (and the result of the application of the evolution strategies), and allows the possibility to undo an operation to come back to the previous state of an ontology.

- **Undo/Redo Facility**
  SHOE does not provide a facility that allows users to reverse the effects of the changes that were performed.

SHOE was developed in 1996 and was one of the first approaches to associate semantic description to web pages following a philosophy similar to JavaScript in the sense that the semantic descriptions were codified using a HTML extended language and embedded in the web pages. Also, the authors of SHOE were one of the first to focus their attention on the problem of versioning and reusing ontologies. SHOE includes an
important capacity for version support where essential tasks like identification of versions of an ontology, and specification of relations between versions, are well defined. On the other hand, SHOE does not support the definition of evolution strategies and mechanisms for decision support in versioning and evolution.

### 3.3.2.3 Ontoview (also known as RDFDiff)

Ontoview [Klein et al., 2002a] is a web-based system developed jointly by VU Amsterdam and Ontotext Lab that provides basic functionality for managing versions of ontologies, and it was specifically designed to compare versions of ontologies in RDF-based languages and to highlight the differences between them. Ontoview was strongly influenced by CVS (Concurrent Versioning System, [Berliner, 1990]). In fact, the initial implementation was based on CVS, but finally the Ontoview platform evolved to support Jena Semantic Web Toolkit [McBride, 2001] and Sesame RDF repository [Broekstra et al., 2002].

There are two methods of importing new ontologies (or new versions of an ontology) and change specifications for an ontology in Ontoview. In the first one, the user introduces the URL where the ontology or change that he/she wants to incorporate is stored. In the second one, the user specifies the physical path (within the local file system) where the ontology is stored. In both cases the user has to provide some information for the identification of the version, like the date when it was imported or the user-id of the person who was doing this. This user has to define in which part of the hierarchical structure of the Ontoview system will be stored the ontology/change that he/she is importing. The unique identifier that distinguishes the version of an ontology can be generated automatically in the form of a URI or can be created by the user (and he/she has to guarantee the uniqueness).

Ontoview compares versions of ontologies at the conceptual level, showing which definitions of ontological concepts or properties have changed and what kind of change occurred (additions, deletions or definition changes).

To compare two ontologies, the tool follows the next algorithm [Klein et al., 2002a]:

1. The two ontologies described in an RDF based language (like DAML+OIL) are parsed into RDF triples, which results in a set of small graphs.
2. The next step is to define a correspondence of triples of the two ontologies and try to detect added or deleted statements.
3. Then the sets of triples are evaluated using a number of rules that codify different change operations. The algorithm starts with basic changes and then looks for more complex operations following a refinement process. The rules specified in a particular RDF-based language follow the next pattern [Klein, 2004]:

---

22 Ontoview was devised to be independent of the ontology level, but the current implementation only supports RDF based ontology languages. Via plug-ins the system can be extended to support other kinds of ontology languages.
IF exist:old
  <A, Y, Z>*
exist:new
  <X, Y, Z>*
not-exist:new
  <X, Y, Z>*
THEN change-type A

4. The process finishes with the visualization of the changes that were detected using a highlight schema representation. An example of the visualization of the results is included in Figures 3.3.2.3.1 and 3.3.2.3.2:

The set of rules mentioned before are specific for any RDF-based ontology language and codified in the semantic for this language.

For example, the following DAML+OIL definition of a class “Person” [Klein, 2004]:

```xml
<daml:Class rdf:ID="Person">
  <rdfs:subClassOf rdf:resource="#Animal"/>
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#hasParent"/>
      <daml:toClass rdf:resource="#Person"/>
    </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>
```

It is translated by Ontoview in the next set of RDF statements (first step of the algorithm that was described before):

```
Person    rdf:type    daml:Class
Person    rdfs:subClassOf   Animal
Person    rdfs:subClassOf   anon-resource
anon-resource  rdf:type    daml:Restriction
anon-resource  daml:onProperty   hasParent
anon-resource       daml:toClass   Person
```

The second step in Ontoview’s algorithm for comparing versions of an ontology is to define correspondences between the RDF statements of the two versions that were generated in the previous step [Klein, 2004a]:
In the third step, the triples are evaluated using the rules that were introduced before. An example of a rule for specifying a change in the property type can be expressed as follows [Klein, 2004]:

\[
\text{IF exist:old}
\begin{align*}
\text{<X, rdf:type, rdf:#Property>} \\
\text{<X, rdf:type, daml:#TransitiveProperty>}
\end{align*}
\text{exist:new}
\begin{align*}
\text{<X, rdf:type, rdf:#Property>} \\
\text{not-exist:new}
\end{align*}
\text{THEN Unset_Transitivity}
\]

For the last step, an example of the visualization of the results is included in Figures 3.3.2.3.1 and 3.3.2.3.2.
Figure 3.3.2.3.1: Example of Visualization of the Comparison of Two Versions of an Ontology (http://test.ontoview.org/index.htm)

Figure 3.3.2.3.2: Example of Visualization of the Comparison of Two Versions of an Ontology (http://test.ontoview.org/index.htm)
Following the schema of previous sections, this section includes an analysis and comparison of versioning and evolution capabilities in Ontoview with respect to the requirements that were proposed for the DIP project:

- **Version identification**
  OntoView provides two methods of persistent and unique identification of web-based ontologies. First, the system can guarantee the uniqueness and persistence of namespaces that start with “http://ontoview.org/”, because the system is located at the domain ontoview.org. Second, it is possible to store ontologies with arbitrary namespaces that are provided by the user (the user is responsible in this case for guaranteeing uniqueness) [Klein et al., 2002a].

- **Version storage**
  The different versions of an ontology are stored in independent locations inside the hierarchy structure for ontologies of the Ontoview system.

- **Data Accessibility**
  Ontoview is a diff tool that compares versions of an ontology. There are no descriptions in the literature about support for data accessibility features (maintaining access to instance data from different versions of the ontology)\(^{23}\).

- **Formal Change Representation**
  As already described, Ontoview provides a formal representation of the changes between versions of an ontology using a number of rules that codify different change operations. The rules are specified in a particular RDF-based language and very dependent on the ontology language that was used to represent the versions of an ontology. Therefore, there is a set of rules for any of the languages supported by Ontoview.

- **Consistent reasoning**
  In [Klein et al., 2002a] the authors mentioned the possibility of using FaCT ([Horrocks, 1998], [Bechhofer et al., 1999]) as an external reasoner to check consistency, but there is no explicit statement in the bibliography that an external reasoner was integrated with Ontoview\(^{24}\).

- **Data Transformation**
  Ontoview does not provide tools to adapt the data to a new version of an ontology.

- **Security support**
  No security mechanisms are described by the authors of the tool. Ontoview runs as a web application server, so the server should include general security mechanisms, but it would be also interesting to have a control module that verifies the changes proposed by the users.

\(^{23}\) Confirmed by Atanas Kyriakov (OntoText) one of the participants in the OntoKnowledge project (http://www.ontoknowledge.org/) and one of the co-authors of Ontoview.

\(^{24}\) Also confirmed by Atanas Kyriakov (OntoText). Ontoview does not provide this capacity.
• Definition of Evolution Strategies
Ontoview is a tool for managing versions and does not provide mechanisms to encapsulate certain policies for evolution with respect to the user’s demands.

• Editing Support
Ontoview provides a simple mechanism that shows the user the consequences of the application of a set of changes in an ontology. This mechanism helps the user who is editing the ontology, and the evaluation of these consequences creates a new set of changes that try to preserve consistency in the rest of the new ontology version. This new set of complementary changes can influence the specification of a set of evolution strategies.

• Undo/Redo facility
No undo/redo facility is supported to allow users to reverse the effects of the changes that were performed.

Ontoview [Klein et al., 2002a] is a web-based system development of which started as a joint project of VU Amsterdam25 and OntoText Lab26. It provides basic functionality to manage versions of ontologies, and it was specifically designed to compare versions of ontologies in RDF-based languages27 and to highlight the differences between them. Several important features for versioning are supported by this tool like versioning identification, version storage and formal change representation. Ontoview was the testing platform used by Michel Klein to evaluate some of the theories that he included in his PhD thesis [Klein, 2004]. Klein’s thesis is one of the references for this document.

3.3.2.4 PROMPT Suite tools, an Integrated Approach
The PROMPT suite consists of a set of tools that have had an important impact on the area of merging, alignment and versioning ontologies. A relevant result of this development is the definition of a global strategy that aims to take advantage of the synergies that generated the combination of tools that in the past were considered independent. PROMPT suite [Noy and Musen, 2003a] includes an ontology merging tool (iPROMPT, formerly known as PROMPT [Noy and Musen, 2000]), an ontology tool for finding additional points of similarity between ontologies for other tools like iPROMPT (AnchorPROMPT, [Noy and Musen, 2001]), an ontology versioning tool (PROMPTDiff, [Noy and Musen, 2002]), and a tool for factoring out semantically complete sub-ontologies (PROMPTFactor, [Noy and Musen, 2003a]). The work of Natasha Noy and colleagues proves that in multiple ontology management, tasks like looking for differences between versions of an ontology or looking for similarities between two ontologies in a merging process are closely interrelated and share several

25 http://www.vu.nl/
26 http://www.ontotext.com/
27 Ontoview was devised to be independent of the ontology level, but the current implementation only support RDF based ontology languages. Via plug-ins the system can be extended to support other kind of ontology languages.
components and heuristics (see Figure 3.3.2.4.1). So tools for supporting some of the
tasks in the context of multiple ontology management can benefit greatly from their
integration with others [Noy and Musen, 2003].

The key components of the PROMPT suite have been developed as extensions (plug-
is) of the Protégé 2000 ontology development environment [Gennari et al., 2002]. We
can distinguish the following components:

- iPROMPT is an interactive ontology merging tool, which helps users in the
  ontology merging task by providing suggestions about which elements can be
  merged, by identifying inconsistencies and potential problems and suggesting
  possible strategies to resolve these problems and inconsistencies.

- AnchorPROMPT extends the performances of tools like iPROMPT or Chimaera
  ([McGuinness et al., 2000]) determining additional points of similarity between
  ontologies that are not identified by iPROMPT.

- PROMPTDiff compares two version of an ontology and identifies structural
  differences between different versions of the same ontology.

- PROMPTFactor is a tool that enables users to create a new ontology, factoring
  out of a part of an existing ontology. In this process, the tool guarantees that the
  terms of the resulting sub-ontology are well-defined (for instance, every concept
  of the sub-ontology includes the appropriate super-concepts/sub-concepts
  required for its specification).

One of the major contributions of the development of the PROMPT suite was the
identification of an important overlap in the functionality of its tools and the
implementation of an integrated approach where all these tools benefit from each other. For instance, some of the components that were originally created for the interface of iPROMPT were re-used in the implementation of the interfaces of the other tools of the suite. In addition, the initial sets of related terms between two ontologies that AnchorPROMPT requires as a start point for a more in depth analysis of similarities can be provided by iPROMPT. In return, AnchorPROMPT can supply an additional set of related terms that can be used by iPROMPT to improve the results of the merging process. A final example of this integrated approach can be found in the design of PROMPTDiff and iPROMPT. PROMPTDiff uses some of the heuristics that were initially developed in iPROMPT for comparison of concept names, slots attached to concepts, domains and range of slots and so on.

As mentioned above, iPROMPT [Noy and Musen, 2000] is an interactive tool implemented as an extension of Protégé 2000. iPROMPT guides users in the process of merging two ontologies (see an example of the user in interface in figure 2). The tool was originally developed to handle ontologies specified in OKBC [Chaudhri et al., 1998], but currently there are substantial efforts underway to adapt the tool in order to support the OWL [Dean and Schreiber, 2004] ontology language.28 The central element of iPROMPT is the algorithm that defines a set of steps for the interactive merging process, see also Figure 3.3.2.4.2. The first step is to identify potential merge candidates based on class-name similarities. The result is presented to the user as a list of potential merge operations. The second step is initiated by the user who selects one of the

28 Based on personal correspondence with Natasha Noy, 19-05-2004
suggested operations from the list or specifies the operation directly. The system performs the requested action and automatically executes additional changes derived from the action. It then makes a new list of suggested actions for the user based on the new structure of the ontology, determines conflicts introduced by the last action, finds possible solutions to these conflicts and displays these to the user.

Initially, PROMPT identified a set of ontology-merging operations (merge classes, merge slots, merge bindings between a slot and a class, etc.) and a set of possible conflicts for these operations (name conflicts, dangling references, redundancy in the class hierarchy and slot-value restrictions that violate class inheritance). These lists of ontology-merging operations and possible conflict operations have been extended by the authors of the tool as part of an evolution process in the design of the system.

The goal of AnchorPROMPT [Noy and Musen, 2001] is to augment the results of methods that analyze only local context in ontology structures (like Chimaera [McGuinness et al., 2000] or iPROMPT [Noy and Musen, 2000]) by finding additional possible points of similarity between ontologies. To do this, AnchorPROMPT requires that the other tool or the user provides an initial set of related terms. Following a graph perspective, the tool establishes a set of paths that connect the terms of an ontology that are related with the terms of another one. The algorithm takes two pairs of related terms as input and analyses the elements that are included in the path that connect the elements of the same ontology with the elements of the equivalence path of the other ontology. So, we have two paths (one for each ontology) and the terms that compound these paths. The analysis looks for terms along the paths that might be similar to the terms of the other path, which belongs to the other ontology, assuming that the elements of those paths are often similar as well. These new potentially related terms the algorithm discovers are marked with a similarity score that can be modified during the evaluation of other paths in which these terms are also involved. Terms with high similar scores will be presented to the user to improve the set of possible suggestions in, for example, a merging process in iPROMPT.
If the two ontologies that we compare present important differences in the number of levels of their hierarchy or in the number of relations between classes, the algorithm does not work well [Noy and Musen, 2001].

The third element of the suite is PROMPTDiff [Noy and Musen, 2003], which is used to compare the structure of two versions of a particular ontology and which identifies the frames (i.e. classes, slots or instances) that have no changes, frames with only changes in their properties, and frames that have also changed in other parts of their definitions. The name of the tool, PROMPTDiff, is influenced by tools like CVS [Fogel and Bar, 2001], which is a version control system that is used to maintain the history of program source code files. This tool includes facilities to discover changes between versions of a document (finding a diff).

The PROMPTDiff [Noy and Musen, 2002] algorithm combines an extensible set of heuristic matchers and a refinement evaluation process that runs all the matchers and combines their results in a table that is presented to the user following the visualization style of the PROMPT suite. The algorithm distinguishes between three kinds of mismatches of two versions of a frame: unchanged (nothing has changed in the definition), isomorphic (the frames have slots, and facet values are images of each other but not identical) and changed (the frames have slots or facet values that are not images of each other).

As we have outlined above, there is a lot of synergy between PROMPTDiff and iPROMPT because they share many heuristic matchers [Noy and Musen, 2002] that were originally defined for iPROMPT. For example, both tools share the heuristic matcher that detects pairs of classes, properties or instances that are the same because the name and the type are the same.

The last element of the PROMPT suite we will describe here is the tool PROMPTFactor [Noy and Musen, 2003a] which allows users to extract from a larger ontology the elements that the user is interested in, in a way that also copies all the terms required for preserving the semantics of the descriptions. The authors of the tool call this process “factoring sub-ontologies”.

PROMPTFactor can be used as an autonomous tool (without the participation of the user in the extraction process) that extracts the sub-ontology that can describe some terms previously selected by the user, or it can be used as an interactive tool in a similar way as iPROMPT, where the user selects the terms that he/she wants to incorporate in the sub-ontology. The tool then looks for possible inconsistencies, caused by incorporating these terms in the sub-ontology. The inconsistencies are presented to the user along with possible strategies for resolving these inconsistencies. Creating the sub-ontology is an iterative/interactive process, in which the user is guided by suggestions from the tool.

During the analysis of the PROMPT suite, we concluded that the tool has some limitations in the area of ontology versioning and evolution. We present a summary of some of the most relevant conclusions of our study (some of them were confirmed by Natasha Noy29):

- Version identification

---

PROMPTDiff can find difference between ontologies but it does not mean that there is explicit support for versioning. For example, PROMPTDiff does not allow the user to identify versions or to indicate that there is a versioning relationship between ontologies.

- **Version storage**
  Following the discussion introduced in the previous point, the user has to find a way to manage different versions of an ontology and to store these in different locations, identifying that a particular ontology is a version of other ontology.

- **Data Accessibility**
  PROMPT suite does not provide mechanisms to maintain access to instance data from different versions of the ontology.

- **Formal Change Representation**
  PROMPTDiff only detects differences between two versions using a structural diff. In [Klein, 2004], we can find several complementary alternatives (change logs, conceptual relations and transformation sets) that can give us a richer description of the changes that the original ontology has undergone. The description of the differences between two versions of an ontology that PROMPTDiff offers is limited. For this reason, Klein extended PROMPTDiff to support richer semantic descriptions of changes. He introduced a more complex classification of type of changes (implicitly-changed, directly-changed, changed, isomorphic and unchanged, see [Klein, 2004]) and provides a high-level description of the changes based on the idea of minimal transformation set and on an ontology of changes (again, see [Klein, 2004]).

- **Consistent reasoning**
  The new Protégé OWL Plug-In ([http://protege.stanford.edu/plugins/owl/](http://protege.stanford.edu/plugins/owl/), [Knublauch et al., 2004]) provides direct access to external DL reasoners like Racer [Haarslev and Moeller, 2003] to check inconsistencies in the ontology definitions. The current user interface supports two types of DL reasoning: Consistency checking and classification (subsumption). Other types of reasoning, such as instance checking, are work in progress.

- **Data Transformation**
  This feature is not supported by PROMPT suite.

- **Security support**
  No security mechanisms are described by the authors of the suite. The set of tools are running as Protégé plug-ins, and Protégé includes security mechanisms that limit the edition capabilities of the users.

- **Definition of Evolution Strategies**
  PROMPTDiff does not provide evolution support and particularly evolution strategies like those in [Stojanovic et al., 2002]. The user has to figure out unaided how to create a new version of an ontology. There are no pre-defined evolution strategies (for example if a user eliminates a class, the system links the subclasses to the super class of the deleted class) where the system could help the user to maintain the coherence of the system after performing some changes.
• Editing Support
  There is no decision support component that could show the user the possible implications of a change in the ontology (and the result of the application of the evolution strategies).

• Undo/Redo facility
  Protégé provides an undo facility that allows users to reverse the effects of the changes that were performed\(^30\).

The PROMPT suite is a set of tools that covers many aspects in the areas of merging, alignment and versioning ontologies, but the versioning module should be improved in the line with what Michel Klein proposed in his PhD thesis ([Klein, 2004]). It also requires an extension with a proper versioning management module (version identification and version storage). In addition no evolution support is provided.

3.3.2.5 KAON Support for Versioning and Evolution

The KArlsruhe ONtology and Semantic Web tool suite ([Gabel et al., 2004] [Volz et al., 2003] [Motik et al., 2002] and http://kaon.semanticweb.org/) is an Open Source ontology management infrastructure that it was mainly developed by members of AIFB at University of Karlsruhe and the FZI (Research Center for Information Technologies) at Karlsruhe. KAON provides a wide range of functionalities for creation, retrieval, maintenance, mapping and evolution of ontologies [Gabel et al., 2004] (see Figure 3.3.2.5.1 for a general overview of KAON).

![Figure 3.3.2.5.1: General Overview of KAON [Gabel et al., 2004]](image)

---
The modules of KAON are grouped into two main parts: the Front-End defines how the suite interact with the users, and the Core implements the functionality to manage local and remote ontology repositories and provides a set of interfaces to allow access to the system. The Front-End can be decomposed into two main modules: KAON Workbench and KAON Portal. The first one provides a graphical environment for ontology management through the OI-Modeler (a graphical ontology editor) and Open Registry (a mechanism for registering and searching features in a distributed ontologies context). The KAON Portal is a tool that provides several features for the generation of multilingual ontology-based Web portals [Gabel et al., 2004].

The core of KAON provides two ontology APIs (KAON API and RDF API) that define a set of interfaces that allow external applications to access the functionalities implemented by the system. The KAON API provides programmatic access to ontologies and instances independent of the physical storage mechanism. On the other hand, The RDF API provides programmatic access to RDF models. It features proprietary means for modularization, fast and efficient RDF parser and serializer as well as transactional access [Motik et al., 2002]. Currently three different implementations of the KAON API and the RDF API are available: the Engineering Server, an ontology server with a scalable database representation for ontologies; the RDF Server for storing and accessing RDF models; and the APIonRDF, a main-memory implementation of the KAON API on the RDF API [Gabel et al., 2004].

OI-Modeler and KAON API provide the evolution facilities of the suite, and it is basically the implementation of the proposal for evolution process and strategies outlined in previous sections and based in the work of Stojanovic and colleagues (see [Maedche et al., 2003], [Maedche et al., 2003a], [Stojanovic et al., 2003] and [Stojanovic et al. 2002]). OI-Modeler allows users to set up preferences for Ontology Evolution Strategies (see Figure 3.3.2.5.2), presents evolution details to the user for approval (the change the users performed can generate additional changes based on the defined evolution strategies in order to maintain the consistency of the ontology, see Figure 3.3.2.5.3), and executes Undo/Redo operations. On the other hand, KAON API implements the evolution functionality for computing sequences of additional changes when it is necessary to maintain the consistency of an ontology after performing a modification, and also it provides the necessary interfaces to provide access to this functionality from external applications [Gabel et al., 2004].
Following the schema of previous sections, this section includes an analysis and comparison of versioning and evolution capabilities in KAON with respect to the requirements that were proposed for the DIP project:
• Version identification
  This feature is not supported by KAON.

• Version storage
  This feature is not supported by KAON.

• Data Accessibility
  This feature is not supported by KAON.

• Formal Change Representation
  KAON uses a change log for tracking the modifications in an ontology. When
  the system has to perform a change, the evolution mechanism generates the
  possible complementary changes in order to maintain the consistency of the
  ontology (or also to undo a change or a group of changes). These groups of
  changes are processed to generate a more compact specification of changes
  through the transformation of groups of simple changes into complex changes.
  Stojanovic and colleagues (see [Maedche et al., 2003], [Maedche et al., 2003a])
  argued that the diff approaches (including (minimal) transformation set) are very
  time-consuming, and change log can be more appropriate for large-scale
  ontologies. On the other hand, Klein ([Klein, 2004]) claims that a change log is
  not always available and the granularity description of the changes can be less
  optimal that using the (minimal) transformation set.

• Consistent reasoning
  KAON currently allows connection to DL engines, which support the DIG
  [Bechhofer et al., 1999] interface and several rule-based inference engines such
  as OntoBroker [Decker et al., 1999] and XSB [Sagonas et al., 1994]. These
  reasoners provide general consistency or satisfiability checking mechanisms that
  could be used to verify that the modifications made to a new version of an
  ontology do not generate inconsistencies with the previous specifications that
  were not changed.

• Data Transformation
  This feature is not supported by KAON31.

• Security support
  This is limited to the ability to support verification and authorization of the
  changes that are applied to an ontology.

• Definition of Evolution Strategies
  KAON includes an evolution mechanism that encapsulates certain policies for
  evolution with respect to the user’s demands. The evolution strategy tries to
  automate the generation of a set of additional changes that preserves the
  consistency of an ontology after performing a certain type of change.

31 Private email correspondence with Boris Motik (FZI) 28-06-2004. Boris confirmed the points: Version
identification, Version storage, Data accessibility and Data transformation.
• Editing Support
KAON shows what could be the consequences of the application of the changes for the new ontology version and shows these potential consequences to the user who requests the changes. The evaluation of these consequences creates a new set of changes which aim to preserve the consistency in the rest of the new ontology version. This new set of complementary changes can influence the specification of a set of evolution strategies.

• Undo/Redo facility
KAON provides an undo/redo facility that allows users to reverse the effects of the changes that were performed.

The *Karlsruhe Ontology and Semantic Web tool suite* ([Gabel et al., 2004] [Volz et al., 2003] [Motik et al. 2002] and [http://kaon.semanticweb.org/]) provides a broad support for evolution features. As we mentioned before, the evolution module of KAON is the implementation of the evolution framework proposed by Ljiljana Stojanovic and colleagues and presented in previous sections. On the other hand, ontology versioning features are not supported.

3.4 Extending Library Systems to Support Concurrent Access and Workgroup Facilities in Distributed Environment

The definition of Library Systems and the evaluation of several tools were presented in [Ding and Fensel, 2001]. The facilities that are included in Library Systems are very useful for the necessities of DIP, but the original idea should be extended to support new features: Concurrent access, workgroup/collaboration facilities, distributed environment or change management. The next sections review some of the technologies that Library Systems should include to fulfil the requirements of the DIP project.

3.4.1 Overview

The concept of Ontology Library Systems was identified and defined in [Ding and Fensel, 2001] in the following way (see also Figure 3.4.1.1):

“An Ontology library system is a library system that offers various functions for managing, adapting and standardizing groups of ontologies. It should fulfill the needs for re-use of ontologies. In this sense, an ontology library system should be easily accessible and offer efficient support for re-using existing relevant ontologies and standardizing them based on upper-level ontologies and ontology representation languages. For this reason, an ontology library system will, at the very least, feature a functional infrastructure to store and maintain ontologies, an uncomplicated adapting environment for editing, searching and reasoning ontologies, and strong standardization support by providing upper-level ontologies and standard ontology representation languages.”
Visualization and Browsing

Visualization and Browsing are two important elements in Library Systems because they define an important part of the interaction with the users. We will introduce a brief description of several techniques that simplify the visualization of ontologies using graphical interfaces. An important part of this analysis is based in the MSc Thesis of Roman Korf at the University of Edinburgh (see [Korf, 2003]).

One of the most common approaches is to employ a tree-structure that visualizes the terms (concepts and classes) of an ontology using a hierarchical graphical representation of nodes and edges where edges define the relationships between the pairs of nodes [Korf, 2003]. A tree has exactly one root node, which is the highest node in the hierarchy. Each node can be reached from the root node, and all nodes can have several children but just one parent node. Consequently there is just one path to each node [Korf, 2003]. This structure has two important limitations. The first one is that terms with multiple parents have to be represented using several nodes or in other words several nodes in different locations of the tree can characterize the same term. Therefore, from one concept we can visualize all the children but not all the parents directly [Korf, 2003]. The second limitation is related to the fact that the information that can be directly displayed for each concept is very poor. Usually the name of the concept is displayed or sometimes a unique identifier. All the other properties cannot be displayed easily if we do not want to sacrifice the visualization scope [Korf, 2003].
Figure 3.4.1.1: shows the (a) tree and (b) graph visualisation of concepts from the GO ontology (http://www.ontologos.org/IFF/Ontologies/Gene.html) with the cellular-components. In (a) the tree visualisation the single concept *vacuolar membrane* has to be represented by two nodes since in a tree a node can have just one direct parent. In (b), the graph visualisation, multiple parents for the concept *vacuolar membrane* presents no problem [Korf, 2003].

The trees can offer several techniques to overcome the limitations of the size of the screens when it is necessary to visualize large, complex ontologies. One possibility is scrolling where the user can move through the ontology and make visible some parts of the tree while other parts will be hidden. Another approach is to expand and collapse sub trees or specific sub trees. Using this technique, it is possible to reduce significantly the visualization area and to hide some of the complexity of the tree. Finally, to hide nodes with specific properties can be another way to reduce complexity [Korf, 2003]. One of the most representative examples of tools that use tree visualization for ontologies is Protégé (see Protégé 2000 User Guide, http://protege.stanford.edu/useit.html, and figure 3.4.1.2).
In recent years, researches have been looking for alternatives to trees. Several proposals were oriented to representing ontologies using graph structures. Like trees, graphs represent the terms of ontologies using nodes and edges. In contrast to trees, nodes in graphs can have more than one parent node. As a result of this characteristic, it is now possible to see from one concept all the children and all the parents. In comparison with tree structures, as we go more deeply into the structure, the visualization gets very complex very quickly, and this is a particularly negative feature in domains where we have to deal with large-scale ontologies. The Brain\textsuperscript{33} is one of the best examples of a graph visualization tool, as the reader can see in figure 3.4.1.3.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{brain.png}
\caption{The Brain}
\end{figure}

\textsuperscript{32} http://protege.stanford.edu/useit.html
\textsuperscript{33} http://www.thebrain.com/
Based on the two previous approaches, trees and graphs, new proposals have come up aimed at minimizing the limitations already outlined. Cluster Map [Fluit et al., 2002], a visualization component of the tool Spectacle\(^{34}\), is one of these alternatives that tries to improve the expressivity of the graph structures. The classes are represented by big spheres with an attached label with the name and the cardinality. Edges connect classes and point from specific to generic class. Classes that are semantically related (because they share many instances) are also geometrically closer. Unfortunately, this approach only fits well with light-weight ontologies. An example of Cluster Maps can be found in Figure 3.4.1.4.

A promising approach, called hyperbolic tree [Lamping et al., 1995], was developed in 1995 in Xerox Parc Laboratories and commercialized by Inxight\(^{35}\). This technique was specifically defined for displaying large tree structures. The current node is visualized enlarged while peripheral nodes are displayed smaller but still provide the context for the current node. A geometric transformation into the hyperbolic plane and then into a circle is used for this purpose. Figure 3.4.1.5\(^{36}\) shows an example of a hyperbolic tree, and Figure 3.4.1.6 does the same but with a concrete implementation called OntoRama\(^{37}\) [Eklund et al., 2004]. This interface proposed a mixed solution that combines a hyperbolic tree with a tree visualization structure.

\(^{34}\) [http://www.administrator.nl]

\(^{35}\) [http://www.inxight.com]

\(^{36}\) [http://www.loria.fr/~bouthier/images/htKlein.jpg]

\(^{37}\) [http://www.ontorama.org/index.html]
Figure 3.4.1.5: Example of Hyperbolic Tree

Figure 3.4.1.6: Example of Hyperbolic Tree in OntoRama

38 http://www.loria.fr/~bouthier/images/htKlein.jpg
39 http://www.ontorama.org/index.html
The ontology graph view of KAON (see Figure 3.4.1.7) is based on TouchGraph\textsuperscript{40} — a popular general-purpose hyperbolic tree visualisation library. The KIM semantic annotation platform uses Touch Graph for instance-level visualisation, as presented on Figure 3.4.1.8; demo available at \url{http://dell.sirma.bg/kim/graph/Graph.jsp}.

\textsuperscript{40} \url{http://sourceforge.net/projects/touchgraph/}
\textsuperscript{41} \url{http://km.aifb.uni-karlsruhe.de/kaon2/Members/hora/User/index.html}
Finally, a very simple approach is known as Node-Based visualization, and it is radically opposite to previous proposals. In node-based visualization, only the information of a concept (properties and relations) is displayed at once using usually a document-style manner. The navigation in the ontology can be done using hyperlinks to the relatives and properties following patterns similar to JavaDoc\textsuperscript{42}. The main disadvantage of this strategy is that the structure of the ontology cannot be displayed. Figure 3.4.1.9 illustrates this visualization technique. The example was taken using Ontolingua [Farquhar et al., 1997].

![Node-Based Example from Ontolingua Tutorial\textsuperscript{43}](image-url)

Similar hypertext-based approaches are also adapted in Cyc (the Open Cyc KB Browser, \url{http://www.opencyc.org/}), the Ontosaurus Loom Web Browser (\url{http://www.isi.edu/isd/ontosaurus.html}), KIM (the KIM Explorer for entities, \url{http://dell.sirma.bg/KIM/}), and Protégé (the menu option Project - Generate HTML).

### 3.4.1.2 Concurrent Access

As emphasized before, one of the main goals in Distributed Ontology Management is to provide concurrent access facilities to enable collaborative work on a single (or multiple) ontology model instance. These multi-user capabilities permit the development of an ontology (or several different ontologies) in a distributed setting, e.g. by a group of ontology engineers or domain experts who are spread across several locations. Concurrent Access (the ability to support multiple operations like editing or

\textsuperscript{42} \url{http://java.sun.com/j2se/javadoc/}

\textsuperscript{43} \url{http://www.ksl.stanford.edu/software/ontolingua/}
retrieval at the same time) and collaboration features (the capability to offer to the users a set of workgroup facilities) are two of the most significant aspects in multi-user environments. In this section, the concurrent access capabilities and some of the most relevant multi-user ontology tools are analyzed. Particularly we are interested in strategies for concurrent editing (multiple users editing the same ontology), and locking policies (several terms of the ontology that is being edited are locked for a particular user for editing purposes).

The *Karlsruhe Ontology and Semantic Web tool suite* ([Gabel et al., 2004] [Volz et al., 2003] [Motik et al., 2002] and [http://kaon.semanticweb.org/](http://kaon.semanticweb.org/)), an Open Source ontology management infrastructure mainly developed by members of AIFB at University of Karlsruhe and the FZI (Research Center for Information Technologies) at Karlsruhe provides in its infrastructure several features for multi-user concurrent access. As already described in previous sections, the suite includes three different back-end implementations of the KAON API (the set of interfaces that allow external applications to access ontologies and instances independent of the physical storage mechanism): the Main Memory implementation, the RDF Server and the Engineering Server. The last one is a storage mechanism for KAON ontologies, based on relational databases and is responsible for the concurrent access support. The Engineering server maintains for each user a local “copy” of the ontology (ontologies) that is current being edited and assigns a version number for each successive state of this ontology (one for each ontology). It is possible to browse and explore the entire ontology without restrictions, but during the editing process the Engineering Server checks the version of the local copy of the ontology that the client is using. If 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 and is required to update the local copy that he/she is using [Gabel et al., 2004] (see Figure 3.4.1.2.1). No details about how KAON reconciles the different local copies of an ontology is provided in the literature that we reviewed, and there is no description about locking policies of ontology terms during the editing process.

![Figure 3.4.1.2.1: Ontology Versioning Conflict in Concurrent Editing](http://kaon.semanticweb.org/)

OntoEdit [Sure et al., 2002] is an ontology engineering environment that combines methodology-based ontology development with capabilities for collaboration and inference. Like in KAON, OntoEdit maintains for each participant a local copy of the ontology that he/she is editing (see Figure 3.4.1.2.2).
All clients are informed about the modifications that are performed in the ontology that they are editing, and unlike KAON, OntoEdit defines a locking policy to prevent two users from modifying the same term of an ontology, or other potential conflicting editing operations like removing a super-concept of the term that another user is editing at the same time.

To guarantee consistent models, the clients are forced to obtain locks for each resource (e.g. concept, instance, relation) that they want to modify. OntoEdit not only locks the resources that users require but also their sub-concepts and super-concepts. This locking information is communicated to the different participants and shown in each client interface (see Figure 3.4.1.2.3). Cross-signs mark the concepts that are locked by other clients and may not be edited. Bullets mark concepts that may be edited, altered and removed at will. Concepts without additional icons are currently not locked and therefore available for locking by any user [Sure et al., 2002].
The approach of KAON, maintaining multiple local copies of an ontology, one for each user, is the most flexible solution because there are no editing restrictions (if the local copy is the latest version). The philosophy is similar to CVS, where users edit documents concurrently, and from time to time each commits the changes to provide their latest version. The most important problem with this approach is how to reconcile the different local copies of an ontology in a consistent new ontology. We could not find more details about how KAON solves this problem, but we believe that a proper versioning mechanism (see Section 3.3) is required to detect and analyze the changes of the different versions.

The Ontolingua Server uses a notion of users and groups for setting the modification capabilities of the people who access to the system. The server provides support for simultaneous work through group sessions. When a user opens a session, she or he may assign a group ownership to it. This enables any other members of that group to join the session and work simultaneously on the same set of ontologies. A notification mechanism informs each user of the changes that other users have made. Notifications are hyperlinked to the changed definitions and describe changes in terms of basic operations such as add, delete and modify.

### 3.4.1.3 Collaboration in Distributed Environment

In a distributed environment, concurrent access mechanisms are not enough for users who are working together editing an ontology (or several different ontologies). In our opinion these concurrent mechanisms should be complemented with tools that provide collaborative facilities that contribute to generating a virtual environment where people can share ideas, discuss approaches or warn other people about certain operations that are currently executed or will be executed.
In the Groupware market we can find different products that cover many of the aspects that we mentioned before, but unfortunately we cannot identify specific solutions in the ontology area. However there are several ontology tools that provide some of these facilities that we will review them briefly in this section.

After the review of some of the literature concerning this topic, we realized that there is confusion between the concepts collaboration, concurrent access and multi-user support. In our opinion, multi-user means that several participants can access or edit an ontology or groups of ontologies. On the other hand, concurrent access means the ability to support multiple operations (like edition or retrieval) at the same time. And finally, collaboration features refer to the capability of offering to the users a set of workgroup facilities for sharing opinions during the process of creating or maintaining an ontology. We should remember that an ontology is a specification of a shared conceptualization [Gruber, 1993]. “Shared” requires consensus between authors who have to discuss several points of view, and collaboration facilities will facilitate reaching this consensus.

The OWL Plug-in [Knublauch et al., 2004] recently developed for Protégé provides a standard set of ontology tests for various best ontology design practices. It also contains tests against OWL DL compliance, for example to warn the user when OWL Full features such as meta-classes have been used. The ontology test mechanism has also been exploited to implement a simple but powerful “to-do-list” feature. A to-do-list is a proposal of tasks which a user suggests to the other participants in the process of creating and maintaining a particular ontology. For example, if a class has the value “TODO: Add German label” as its value of the owl:versionInfo annotation property, then the ontology test button will display a corresponding to-do item in the list of warnings (see Figure 3.4.1.3.1). This simple mechanism helps coordinate shared ontology design efforts [Knublauch et al., 2004]. Although it is a useful feature, the collaboration facilities of Protégé are not sufficient. For example, in the case of these to-do lists, it is not specified who has to do the modifications and when they should be expected. The interaction between users has to be managed with independent tools (like an instant messenger product) that are not integrated with Protégé and cannot reference directly to the information referred to in the to-do list. The next paragraphs discuss other alternatives that offer better support in this area.

44 Computer Supported Cooperative Work (CSCW) is one of the related scientific fields.
OntoEdit [Sure et al., 2002] provides two tools designed specifically for the specification of requirements in the design of an ontology. OntoKick includes specific features for the collaborative generation of requirements specifications for ontologies, and Mind2Onto is a plug-in for supporting brainstorming and discussion about ontology structures (see Figure 3.4.1.3.2). Also, during the design phase, the participants can store comments (like for example design decisions) in the documentation field of each term of an ontology, and clients are immediately informed of changes that other participants are producing.
KAON ([Gabel et al., 2004] [Volz et al., 2003] [Motik et al. 2002] and http://kaon.semanticweb.org/), in our opinion does not provide real facilities for collaboration as opposed to multi-user (concurrent) support, which is a different concept. As it was presented in Section 3.4.1.2, KAON allows users to work together in editing the same ontology and can warn them about conflicts in the modifications that they propose, but there are no workgroup facilities, for example. The same point arises in relation to Ontolingua, which has a similar notification mechanism. The authors of the system stress the ability of Ontolingua to support group sessions where the users who can work on an ontology are precisely identified and organized in groups.

The last tool that will be reviewed in this section is Tadzebao [Domingue, 1998] an ontology discussion tool which supports asynchronous and synchronous communication facilities. Tadzebao runs over WebOnto [Domingue, 1998], an ontology library system developed by the Knowledge Media Institute of the Open University (UK), designed to support collaborative creation, browsing and editing of ontologies.

The name of Tadzebao is not casual and it is intimately related to the purposes of the designers of this tool. Tadzebao is a Chinese word that literally means “Big Character Poster”. This type of poster was used to support political debate during the Chinese Cultural Revolution where a political argument or ideology would be expressed through the placement of a poster (a tadzebao). Rebuttals to or comments on the initial argument would be expressed by additional posters (tadzebaos) on top of or around the original poster [Domingue, 1998].
Tadzebao is integrated in the architecture of WebOnto and includes two main components: a Tadzebao client which is the front-end of the tool and manages the interaction with the users; and a Tadzebao server that represents the back-end and maintains all the annotations that the clients include in the client-side. Tadzebao client uses the idea of virtual notepad that integrates editing tools, such as text editors and drawing tools, for expressing general ideas about the definition or modification of ontologies. Users can include hand-drawn sketches, GIF images, text comments and ontology components represented in OCML (Motta, 1998). These inputs are automatically copied to the Tadzebao clients that are affected, so the users can follow and participate in the discussion “in real time” (synchronous) or see the result of the discussion when they start the client (asynchronous). An example of a screen snapshot of Tadzebao client is presented in Figure 3.4.1.3.3.

![Figure 3.4.1.3.3: A Screen Snapshot of Tadzebao [Domingue, 1998]](image)

Finally, APEKS [Tennison and Shadbolt, 1998] (Adaptative Presentation Environment for Collaborative Knowledge Structuring), a tool with collaborative facilities for creating personal ontologies, is presented. The approach of APEKS is to allow users to
define their own versions of an ontology, and then apply comparison mechanisms to detect differences between versions and prompt these differences to the users in order to start a discussion and reach a consensus between the different proposals. Users can interact with the system and with each other using a program based in a previous development called MOO [Curtis, 1992] (Multi-user text-based virtual environment). MOO supports synchronous and asynchronous textual communication (no multimedia facilities like in Tadzebao) where the comments of the users are displayed in chronological order. MOO also supports a structured asynchronous communication where the system has a predefined set of questions and answers that can be extended by the users. The system prompts general questions in the beginning and a set of possible answers that the users select and fill with specific data. The answers chosen lead to new questions and new possible answers. In the end a hierarchy of questions and answers is provided that can be used for documentation purposes.

Although collaboration facilities can be interesting in distributed environments like DIP, none of the tools described already are widely exploited. Particularly interesting in our opinion is Tadzebao because it includes several features appropriate for starting the definition of a collaborative environment. Synchronous and asynchronous communication mechanisms and a notepad to annotate ideas (textual or graphical) are very useful features. On the other hand, a better integration with the visual editing tool (in this case WebOnto Client) would be desirable.

3.4.1.4 Data Integration and Neutral Ontology Representation

In the context of ontology management in DIP, the authors of deliverable D2.2 [Kiryakov et al., 2004] identified the following problems related to ontology representation: diversity of ontology languages, no mature implementation of ontology servers, integration with real-world applications and ontology and data modularization mechanism. They define a framework called ORDI (Ontology Representation and Data Integration) that provides data-model and epistemology with minimal commitment to semantics and syntax. ORDI is ontology-language neutral. The epistemology provided is similar in spirit to the OO-language and close in complexity to OWL Lite. A major goal of this model is to provide a sufficient coverage and integration with relational databases that are currently storing an important amount of information that is necessary in Web Services and Semantic Web Services areas.

ORDI can be the middleware layer of DIP (like an Interlingua) that can solve the problems of interoperability between several repositories that use different ontology representation languages.

Minimal Semantic Commitments and the coexistence of this approach with the work that it is doing in work-package 1 (D 1.1, D1.3, D1.4 and D1.7) are two points of discussion. The minimal semantic commitments of ORDI try to define some constraints on the interpretation of the epistemological primitives. Sets of axioms are defined that provide constraints for the interpretation of statements and patterns of statements in the

---

45 **Data-model** determines the basic data-structures used to carry knowledge that is represented in a language [Kiryakov et al., 2004].

46 **Epistemology** defines the language at a conceptual level. It determines what paradigm and constructs are used to represent knowledge in this language [Kiryakov et al., 2004].
data model [Kiryakov et al., 2004]. This limited expressivity does not compromise the capacity of a system to provide access to several sources of information represented using different formalisms, but the system is limited in interpreting the semantic description of these formalisms (inference capabilities are restricted). On the other hand, this minimal semantic commitment does not mean that it is not possible to extend the semantic expressivity of the model adding new layers with more axioms on top, and in addition, a simple model allows future users to extend it in several directions without the restrictions of a more complex framework.

The other element of discussion is the coexistence between DIP deliverables D2.2 and D2.6 with D1.1, D1.3, D1.4 and D1.7. The diversity of knowledge representation formalisms and the necessity to provide enough expressivity to describe web services and their interactions are two goals that will be tackled in DIP. D2.2 and D2.6 represent a possible approach where the focus is to guarantee interoperability using a neutral intermediate representation layer and providing links with some popular reasoner engines, but the model is not very expressive and it does not provide a specific reasoner for this model. On the other hand, the approach in DIP Work-Package 1 is slightly different because the idea is to combine two specific very well-known formalisms (OWL-DL and F-Logic (LP)\(^{47}\)) in a new model called hybrid reasoning. DLs and F-Logic are currently considered as alternatives for ontology modelling in the Semantic Web (OWL and WSMO\(^{48}\)). D1.1 shows that none of them is sufficient for realizing complex applications. A Hybrid framework is proposed that can overcome these limitations by combining features of these formalisms. This hybrid mechanism will have greater expressivity and will incorporate a specific reasoning engine (developed within the KAON) which will assist in solving some interoperability aspects (like integration of heterogeneous data sources or schema mapping specification).

This Hybrid framework is more expressive than ORDI and provides reasoning capabilities. In contrast, ORDI offers a more flexible model that can provide access to several representation formalisms (not only OWL –DL and F-Logic (LP)), but ORDI offers a limited expressivity (that can be extended) and no reasoning capabilities (links to external reasoners will be provided).

3.4.1.5 Ontology Views (Factoring)
A database view provides a visualization of a portion of the instances of a database. A database view is defined as a query that can be stored and re-used. Particularly in large databases, views are very useful, because they allow users to visualize the instances in a way that simplifies the interpretation of the information. Also Views can be referenced and used in applications as “virtual tables”. Currently the ontology research community is trying to reproduce the idea of views in ontologies following also a query-based approach where an ontology view should be specified as a query using some of the ontology query languages that have been already proposed. However, ontology views are a more ambitious issue because not only the instances are extracted but also the terminological specification. In addition, extract self-contained portions of ontologies are more complex.

\(^{47}\) F-Logic limited to function-free Horn rules.

\(^{48}\) www.wsmo.org
Two approaches are introduced in this section. The first one, implemented within KAON, defines an extension of the query language RQL ([Alexaki et al., 2000a], [Alexaki et al., 2000]) to generate views from RDFS ontologies [Volz et al., 2003a]. Volz and colleagues distinguish between two types of views. Views on classes applied to concepts (classes) returns only unary predicates, and views on properties can be defined using arbitrary queries which return binary predicates. Also they provide a classification of views on classes and examples of syntaxes based on the example of ontology of Figure 3.4.1.5.1:

![Figure 3.4.1.5.1: Example of Simple Ontology in RDFS](image)

- **Selection View**
  A subset of members of a certain class is selected. The following example generates a virtual class “x:Supervisors”, which is populated with those employees who supervise someone [Volz et al., 2003a]:
  ```rql
  CREATE CLASS VIEW x:Supervisors
  ON x:Employee
  USE
  SELECT INSTANCE
  FROM {INSTANCE} x:supervises
  ```

- **Difference View**
  A subset of members of a certain class that are not in another class is selected. The following example defines a virtual class “x:Unemployed-Students” which is populated with all students that have no job [Volz et al., 2003a]:
  ```rql
  CREATE CLASS VIEW x:Unemployed-Students
  ON x:Student
  MINUS x:Employee
  ```

- **Union View**
A set of members of several certain classes is selected. The following example specifies a virtual class “xScientists” which consists of professors as well as PhD-students [Volz et al., 2003a]:

```
CREATE CLASS VIEW x:Scientists
ON x:Professor
UNION x:PhDStudent
```

- Intersection View
  A subset of members of a certain class that are in another class (or other classes) is selected.

The second approach presented in [Noy and Musen, 2004] argue that the uses of extended query languages to generate ontology views like in the previous proposal do not allow users to specify a portion of an ontology that results from a particular traversal of ontology links. Noy and Musen define a Traversal View as “a subset of an ontology that consists of classes and instances on the path of the traversal specified in the view”.

To generate a Traversal view, they propose a method that starts with the selection of a concept that will belong to the Traversal view, a list of relationships (property names) that should be traversed and the maximum distance to traverse along each of the relationships.

The proposal of Noy and Musen is oriented ontology specified in RDF Schema, but they maintain that it can also be applied for OWL ontologies with some limitations. This is an advantage over the previous approach that also is restricted to ensure that the view returns only unary and binary predicates [Volz et al., 2003a]. On the other hand, [Noy and Musen, 2004] do not include an example of query language, but it is implemented in a Protégé plug-in that includes a graphical interface that simplifies the task of formulating the queries (without typing).

### 3.4.2 Evaluation Schema

In this document we propose to extend the original concept of ontology library systems to bring it closer to the requirements that we think necessary to cover in DIP. These requirements are identified and categorized as follows:

**Management.** Ontology Library systems should provide mechanisms to store ontologies in distributed locations as replicas or as dependent structures that create a modular ontology (see [Klein, 2004]). Also these systems have to offer security mechanisms that restrict the access to the information and guarantee the coherence and consistency of the ontologies that are stored. Concurrent access and particularly concurrent editing are two characteristics which are very desirable in a distributed environment that can be complemented with collaborative tools in order to make user work much easier. Another important element in Library Systems is the possibility of reusing ontologies through import-export mechanisms or using techniques to generate sub-ontologies (factoring) or merge/mapping ontologies to expand the description of a particular domain. These are the main elements that we have identified in our analysis:

- Storage
Define how the ontology is physically stored and analyze which components of the system architecture are focused on this aspect. The architecture model (client or server also known as one-tier, client/server or two-tier, client/application server/database server or three-tier, n-tier) can have an important repercussion on the availability of the system and its failure tolerance. Many database servers provide an important range of functionalities related to replication, transactions management, concurrent support or security.

- **Identification**
  Define a unique identifier for each ontology that is stored in the system.

- **Editing**
  Ontology Library Systems should provide mechanisms that guarantee the execution of remote editing tasks and the concurrent editing of the same ontology. Locking policies, replication processes, n-tier architectures or collaborative tools are closely related to the editing mechanisms.

- **Collaboration**
  A collaborative tool is required for setting up a virtual environment where users can communicate with each other to share ideas, discuss approaches, analyze potential changes in real-time and warn users about potentially dangerous operations.

- **Import and reuse**
  In Library Systems, a very important aspect is the ability to re-use ontologies through reference mechanisms (like in KAON and SHOE) or using merging or mapping/alignment techniques. Some of the ontologies that we would like to re-use can be stored in different systems that can use particular knowledge representation languages. Import facilities can allow the system to access these ontologies.

- **Export**
  Similar to import, it is useful to have mechanisms to export ontologies and to be able to do so in several knowledge representation languages.

- **Versioning and Evolution**
  As we noted in previous sections, versioning and evolution mechanisms provide a very useful functionality for Library Systems. They guarantee data accessibility through different versions and facilitate editing tasks. For concurrent editing purposes it is also advantageous to maintain temporal identifiers of the different versions of an ontology that each user generates when those include simultaneous modifications in an ontology. Then the system is able to identify the versions and to reconcile them into a consistent one using versioning mechanisms to recognize the differences.

- **Merging**
  Merging provides a very powerful mechanism for re-using ontologies through creating a new ontology from two or more existing ontologies with overlapping parts.

- **Mapping and alignment**
  Like merging, these techniques are very useful for re-using purposes. The difference here is that the ontologies maintain their independence. If it is not
possible to import the ontologies for a merging process, mapping and alignment allow us to extend the description of our domain in situations were we do not have full access to ontologies. Also it is important to consider the possibility of evolving the originals ontologies, and how to include these changes in a merged ontology or in a mapping specification. Some authors (see for instance [Maedche et al., 2002a]) claim that it is easier to include changes in mapped ontologies than in merged ontologies. Common features for ontology mapping and alignment tools are [de Bruijn et al., 2004]: Mapping identification and specification, Instance transformation, Instance unification and Query rewriting. The main goal of a mapping mechanism is to identify pairs of equivalent terms in two ontologies. Then it is necessary to specify in a formal way how two equivalent terms are related. Instance transformation is an element of the mapping specification that defines how an instance of the source ontology has to be transformed to be an instance of the target ontology. Instance Unification is another component of the mapping specification that identifies instances that refer to the same real-world object. Instance transformation and unification mechanisms are very useful in querying operations. Query rewriting is a technique that evaluates and adapts a query to be able to retrieve information for heterogeneous information sources (like for example a set of mapped ontologies).

- **Factoring**
  Particularly interesting in large scale ontologies, factoring allows users to extract complete sub-ontologies. This mechanism can be used to simplify the visualization or interpretation of one part of our domain. Some authors (see [Volz et al., 2003a] and [Noy and Musen, 2004]) show these sub-ontologies as views in a database.

**Accessibility.** Ontology library systems should facilitate the task of accessing specific elements of the ontologies that are stored. User-friendly environments for searching and browsing ontologies are especially important when working with large-scale ontologies. On the other hand, reasoning mechanisms allow users of the systems to infer new information for an ontology, or a group of ontologies.

- **Searching**
  Query Languages, Graphical Query Tools and Query Store facilities should be present in the searching mechanism of Library Systems. Some users feel more comfortable with query languages because they allow easy re-use of previous queries and in general provide a more flexible and less restricted query construction. On the other hand, Graphical Query Tools are in general more intuitive for non-expert users. Nevertheless, both require facilities to present the results in a graphical way and to store queries for re-use purposes.

- **Browsing**
  Several methods have been proposed for visualization of ontologies or query results. Tree and graph-based visualization are most common, together with a number of hypertext-based approaches. Nevertheless other proposals like hyperbolic trees and mixed solutions should be taken into account.

- **Reasoning** (inference across multiple ontologies)
These mechanisms infer new information from an ontology or group of ontologies. In a distributed environment, reasoning engines should support modular ontologies (the ontology is physically distributed), mapping-alignment of ontologies, and more generally, multiple ontologies.

**Standardization.** Information interchange is one of the core functionalities aimed at in Semantic Web technologies. To achieve this, the DIP Ontology Management Environment has to support existing and emerging standards in ontology languages to allow information interchange and interoperability with other Semantic Web enabled applications.

- **Language**
  
  Library Systems should support formal languages for describing ontologies, and in particularly languages that have a broad support like for example OKBC, RDF (S) or OWL.

- **Upper-level ontologies**
  
  The upper-level ontology models the basic concepts and knowledge that can be re-used in creating new ontologies and in organizing ontology libraries. The ontology library system should be grounded in any existing upper-level ontologies, such as Upper Cyc Ontology, SENSUS, MikroKosmos, the PENNMAN Upper Model, and IEEE SUO upper-layer ontology.

![Ontology Library System Diagram](image)

**Figure 3.4.2.1: Extended Ontology Library System Structure**
3.4.3 Evaluation of Existing Tools

We identified and analyzed several tools that can be categorized as ontology library systems. These include: WebOnto\(^{49}\) (Knowledge Media Institute, Open University, UK), Ontolingua \(^{50}\) (Knowledge Systems Laboratory, Stanford University, USA), DAML Ontology library system\(^{51}\) (DAML, DAPAR, USA), SHOE\(^{52}\) (University of Maryland, USA), Protégé\(^{53}\) (SMI, Stanford University, USA) and KAON (University of Karlsruhe)\(^{54}\). There follows a brief analysis of each of these tools.

3.4.3.1 WebOnto and Tadzebao (KMI)

WebOnto \cite{domingue1998} is an ontology library system developed by the Knowledge Media Institute of the Open University (UK). It is designed to support the collaborative creation, browse and edition of ontologies. It provides a direct manipulation interface displaying ontological expressions and also an ontology discussion tool called Tadzebao, which could support both asynchronous and synchronous discussions on ontologies (see Figure 3.4.3.1.1).


\begin{itemize}
\item \cite{domingue1998}
\end{itemize}
Management

- **Storage.** WebOnto relies on a third-tier-based architecture. WebOnto Server is responsible for storing and maintaining ontologies represented in OCML (Operational Conceptual Model Language, [Domingue, 1998]). Tadzebao Server maintains the notepad library that contains all the user dialogues. LispWeb HTTP Server manages the requests of the clients (WebOnto or Tadzebao). Ontologies stored in the WebOnto are not classified according to some existing categories. Ontologies are divided into small units (for instance, the ontology is the tree-structure of classes, and the small unit is the class and its parents). They are then stored in a specific Module containing name, type, and the names of class parents. This system can draw graphical representations of ontologies based on the modular storage.

- **Identification.** Ontologies stored in the WebOnto library system are identified by their unique names. Even though an ontology is divided into small units, each unit contains the name, type, and the names of the class parents.

- **Editing.** WebOnto provides a WebOnto client that provides some facilities for visualizing and editing ontologies. The tool allows users to add new terms to the ontology, edit current terms and modify terms using a graphical interface.

- **Collaboration.** Tadzebao is designed for discussing the editing and maintainence process of ontologies (see Figure 3.4.3.1.1). It supports both asynchronous and synchronous discussions and allows users to include graphical information (like sketches or pictures).

- **Import and reuse.** OCML is very close to Ontolingua, so ontologies defined using Ontolingua can be easily transformed and imported into WebOnto.

- **Export.** Translating OCML ontologies into Ontolingua is relatively easy. No other export formats are described.

- **Versioning and Evolution.** WebOnto only notes that ontologies can be inherited from ancestor ontologies. No actual versioning support is provided.

- **Merging.** Not defined.

- **Mapping and alignment.** Not defined.

- **Factoring.** Not defined.

Accessibility

- **Searching.** Ontologies are graphically displayed. They can be browsed by using visualization commands, such as viewing a new ontology or inspecting its structure. No direct query interface is provided.

- **Browsing.** Ontologies can only be browsed by using browsing commands.

- **Reasoning** (inference across multiple ontologies). OCML is written in Lisp, and Lisp engines can be used for reasoning, but OCML does not define particular mechanisms for reasoning for multiple ontologies.
Standardization

- **Language.** Ontologies (classes, instances, functions, procedures, or rules) are represented in OCML [Domingue, 1998] only. In other words, no standard representation languages for ontologies are supported.

- **Upper-level ontologies.** WebOnto does not include a 'giant' standard upper-level ontology but has a more fine-grained structure ([Domingue and Motta, 2000], [Motta et al., 2000]). At the top there is the base ontology describing the meta-model of OCML (things such as relations, functions, procedures, classes, instances, slots, etc., similar to SHOE). Below are the imported (with a few modifications) 'simple time' ontologies from the Ontolingua library and ontologies describing organizations, technologies, events and basic common concepts. Figure 3.4.3.1.2 shows the typical ontology inclusion hierarchy in the WebOnto ontology library system.

![Structure of Upper-level Ontology in WebOnto](image)

Figure 3.4.3.1.2: Structure of Upper-level Ontology in WebOnto [Ding and Fensel, 2001]

3.4.3.2 **Ontolingua Ontology Library System (Stanford)**

Ontolingua [Farquhar et al., 1997] was developed in the early nineties at the Knowledge Systems Laboratory of Stanford University. It consists of an Ontology Server, an Application Server that provides editing facilities through Web pages, and a translator from Ontolingua language to other Knowledge Representation Languages (see Figure 3.4.3.2.1). The Ontology Server maintains a repository of ontologies to assist users in the process of generating new ontologies and amending the existing ontologies collaboratively [Farquhar et al, 1997].
Management

- **Storage.** Ontolingua was created using a third-tier-based approach that facilitates ontology construction, use and re-use in a distributed environment. Access to the contents of ontologies is provided via http protocol, a network API and access to information derived from the contents by a general-purpose reasoner. The ontology server works like a database server and can enable distributed ontology repositories for editing, browsing, etc. The ontology server of Ontolingua supports a suite of other services, including configuration management for ontologies, support for ontologies that have components resident on remote servers, and support for an *Ontology-URL* that enables ontologies to be linked to the World Wide Web [Fikes and Farquhar, 1999].

Ontologies stored in Ontolingua are not classified according to some existing categories. Ontology re-use in Ontolingua is supported by a modular structured library based on the following functions: inclusion, polymorphic, refinement, and restriction. Ontologies in this ontology library system are organized based on the lattice theory. Each ontology defines a set of formal terms. Ontologies can include (import from) other ontologies. Terms contained in an ontology are in the namespace of the ontologies that include it. In the lattice, an ontology includes the ontologies under which are indented (see Figure 3.4.3.2.2). It applies the minimization and amortization principles enabling ontology writers to re-use existing ontologies in a flexible, powerful way [Farquhar et al., 1997].

---

55 This figure is a modified version of the original created by Jan Henke (DERI Innsbruck).
Figure 3.4.3.2.2: Part of the Lattice of Ontologies in the Ontology Library System of Ontolingua [Farquhar et al., 1997]

- **Identification.** Each ontology has a name that uniquely distinguishes it from any other ontology.

- **Editing.** Ontolingua features four basic types of pages for the simple interface: the table of contents for the ontology library system, ontology summary pages, frame pages (for classes, relations or instances), and the class browser. Remote distributed groups can use their web browsers to build and maintain ontologies stored at the server. However, concurrent edition is not supported. Ontolingua supports vocabulary translation, which enables ontology builders to specify translation rules declaratively between the vocabulary used in a source ontology and the vocabulary used in a target ontology [Fikes and Farquhar, 1999]. Ontolingua allows users to undo or redo any number of modifications made to the ontology since it was last saved.

- **Collaboration.** As previously mentioned, we understand that no real collaboration facilities are provided.


- **Export.** The same as for the previous one.

- **Versioning and Evolution.** The Ontolingua server does not support any versioning functions.

- **Merging.** The Ontolingua server does not provide merging capabilities.

- **Mapping and alignment.** These features also are not supported.

- **Factoring.** Not supported.

**Accessibility**
• **Searching.** Visualization capabilities in Ontolingua have been superseded by new tools. Basic graphical features for ontology *browsing* and supporting for swift jumps from one term in the ontology to others term using hyperlinks are provided. Ontolingua’s class/subclass browsers can display an entire hierarchy in compact fashion, offering users a swift overview of an ontology. In relation to the query answering for ontologies, Ontolingua includes an *idiom-based retrieval feature* that returns instances of a sentence containing diagrammatic variables from a given ontology. The retrieval feature employs a general purpose reasoner (i.e., theorem prover) and classifier that can be run as a background process to infer and cache sentences that match idioms used by the API and by the translators. The general purpose reasoner developed for the Ontolingua representation language can provide basic reasoning support for ontology services including classification, deriving and catching instances of sentence diagrams to support idiom-based ontology access, ontology testing, and client-side execution [Fikes and Farquhar, 1999]. Ontolingua also provides several tools that allow users to search for terms within ontologies in the library. A user may choose to use wild cards in searching the entire library for terms whose name matches the specified pattern. Context-sensitive searching is also available when the user needs to fill in the name of a term, by for instance, adding a value to a slot. Constraints are used to limit searches in context-sensitive searching. Finally, Ontolingua provides a *Reference ontology* that serves as an index of the ontology repository for class-based retrieval. Users can browse the reference ontology looking for classes of interest in the repository.

• **Browsing.** It is limited to the HTML features available in 1996. A class and its properties, or a class and their instances are the possible visualization choices, a very simple approach known Node-Based visualization.

• **Reasoning** (inference across multiple ontologies). Ontolingua Server currently does not provide much inferential power. It does provide some support for using ontologies. One way to use ontologies developed with the Ontolingua Server is to translate the ontology into the representation language of another system such as CLIPS, LOOM, or Prolog.

**Standardization**

• **Language.** The ontologies are stored primarily in KIF. KIF is a monotonic first-order logic with a simple syntax and some minor extensions to support reasoning about relations. This language provides explicit support for building ontological modules that can be assembled, extended, and refined in a new ontology [Genesereth and Fikes, 1992].

• **Upper-level ontologies.** The public version of CYC upper-level ontology, called HPKB-UPPER-LEVEL with extended material drawn from Pangloss, WordNet, and Penma, is available on the Ontolingua server [Aitken, 1998]. It contains approximately 3000 concepts, English definitions, and a few basic relationships between them. This upper-level ontology aims to maximize re-usability, enabling a greater degree of interoperation among knowledge-based systems by trying to account for all features associated with one event.
3.4.3.3 DAML Ontology Library System

The DAML ontology library system is part of the DARPA Agent Mark-up Language (DAML) Program, which officially started in 1997. The goal of the DAML effort is to develop a language and tools to facilitate the concept of the Semantic Web. The ontology library system contains a catalogue of ontologies developed using DAML (see Figure 3.4.3.3.1)\(^\text{56}\). This catalogue of DAML ontologies is available in XML, HTML, and DAML formats. People can submit new ontologies via the public DAML ontology library system.

![Figure 3.4.3.3.1: Example of an Ontology in the DAML Ontology Library](http://www.cs.man.ac.uk/~Ehorrocks/DAML-OIL/)

**Management**

- **Storage.** The DAML ontology library system is client/server-based. The structure of the stored ontology in this library system includes: ontology uri; ontology id; description; keyword; poc (point of contract): name, organization, email; submitter: name, organization, email; dmoz (open directory category); funder; classes (class names); properties (properties names); and namespaces (for example, Figure 3.4.3.3.1). Ontologies are classified according to Open Directory Category ([www.dmoz.org](http://www.dmoz.org)), which includes arts, business, computers, games, health, home, kids and teens, news, recreation, reference, regional, science, shopping, society, sports, and world. This library also provides a summary of submitted ontologies, sorted by URI, Submission Date, Keyword, Open Directory Category, Class, Property, Funding Source, and Submitting Organization. The DAML ontology library system was not further developed. It

only provides a simple environment for people to submit and browse ontologies in the library. No module storage is considered at this moment.

- **Identification.** Ontologies are identified by the URIs and identifiers.
- **Editing.** No specific editing functions are available. The users have to create the ontology using other tools and then submit their ontology to the DARPA ontology library system.
- **Collaboration.** This feature is also not supported.
- **Import and reuse.** Not supported directly, but tools are available for such as transformations.
- **Export.** The same that before can be also applied here.
- **Versioning and Evolution.** No versioning functions are provided.
- **Merging.** Not supported.
- **Mapping and alignment.** Not supported.
- **Factoring.** Not supported.

**Accessibility**

- **Searching.** This ontology library system offers no specific searching features.
- **Browsing.** The system contains only a catalogue of ontologies in three formats: XML, HTML and DAML. The HTML version can help users search by generated indexing of ontologies on URI, submission date, keyword, open directory category, class, property, namespace used, funding source, and submitting organization. The other two formats support simple browsing only.
- **Reasoning** (inference across multiple ontologies). The literature does not mention that this library system provides and uses a particular reasoner, but several inference engines for DAML+OIL are already available.

**Standardization**

- **Language.** The library system supports language standards for the semantic web, i.e. RDF, RDF Schema and DAML+OIL, but unfortunately it has not been updated for OWL.
- **Upper-level ontologies.** There is no upper-level ontology for the DAML ontology library system.

**3.4.3.4 SHOE Ontology Library System (Maryland)**

SHOE (Simple HTML Ontology Extensions) was developed by the University of Maryland (USA) ([Heflin et al, 1999], see Figure 3.4.3.4.1). SHOE is also the first web-semantics language developed as a mark-up, and has been used for various applications, including for food safety for the US Food and Drug Administration and a military logistics planning system.
• Storage. SHOE’s ontology library system contains lists of ontologies. These ontologies are indexed alphabetically. They are also classified based on the ontology dependency with a clear tree structure. The upper-level ontology (Base Ontology), for instance, forms the root of the tree. The generic ontologies (e.g. Dublin Core Ontology, General Ontology, and Measurement Ontology) form the first branch of the tree. And the specific ontologies (e.g. Beer Ontology, Commerce Ontology, and Personal Ontology) make up the leaves of the tree. Except for the upper-level ontology, each ontology is stored in the standard format, including ontology ID, version, description, contact, revision date, extended ontologies, renames, categories, relationships, constants, inferences, definitions, notes and change history.

• Identification. The unique name becomes the identifier of each ontology.

57 http://www.dublincore.org/
• **Editing.** SHOE does not provide editing features. Users have to edit their ontology somewhere else and submit it to SHOE.

• **Collaboration.** These features are not supported either.

• **Import and reuse.** No tools are provided for importing ontologies, but on the other hand, users can re-use ontologies during the creation of new ontologies. With the appropriate labels it is possible to refer to a specific ontology and to use its concept and relation definitions.

• **Export.** The tool does not include facilities for exporting ontologies to other representation languages.

• **Versioning and Evolution.** SHOE’s versioning scheme is very essential in handling different types of revisions. It maintains each version of ontology as a separate web page and each instance must state the version to which it adheres. Data sources can, therefore, upgrade to the new ontology. To enter a revision in SHOE, the ontology designer copies the original ontology file, assigns it a new version number, then adds or removes elements accordingly. If the revision merely adds ontology elements, then it can be used to form perspectives that semantically subsume the original perspective. Therefore, it can specify that it is compatible with previous versions using the optional BACKWARD-COMPATIBLE-WITH field in the `<ONTOLOGY>` tag [Heflin and Hendler, 2000].

• **Merging.** No merging facilities are mentioned in the literature.

• **Mapping and alignment.** Although no mapping or alignment tool is included, the authors of SHOE suggest the use of mapping rules to specify mapping relations between ontologies. Also they propose three ways to achieve this goal: mapping ontologies, mapping revisions and intersection ontologies. These approaches were already presented in section 3.3.2.2.

• **Factoring.** SHOE does not provided facilities to extract subontologies.

**Accessibility**


• **Browsing.** Browsing of ontologies is not well supported. The alphabetical indexing of ontologies enables users to browse SHOE ontologies.

• **Reasoning** (inference across multiple ontologies). Parka and XSB provide reasoning capabilities for SHOE, and the possibility of including ontologies in other ontologies allows the system to reason across multiple ontologies.

**Standardization**
• **Language.** An ontology is written in SHOE. SHOE is a HTML-based knowledge representation language. It is a superset of HTML, which adds the tags to semantic data. There are two categories for SHOE tags: tags for constructing ontologies and tags for annotating web documents. There is also an XML-based version of SHOE [Heflin et al., 1999].

• **Upper-level ontologies.** The base ontology is the upper-level ontology for SHOE. It becomes the parent ontology for all SHOE ontologies on the web. All other SHOE ontologies extend the base ontology directly or indirectly. The base ontology declares the global data types (string, number, date and truth), ISA hierarchy (entity, SHOE Entity), and relationships (description and name). There is a one-to-one correspondence between a version of SHOE and a version of the base ontology. Thus, the version of the base ontology reflects the version of SHOE.

### 3.4.3.5 Protégé (Stanford)

Protégé-2000 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. It is an open source, standalone application that provides a graphical ontology editing environment (see Figure 3.4.3.5.1) and an extensible architecture for the creation of customized knowledge-based tools. Its knowledge model is OKBC-compatible.

Its component-based architecture enables system builders to add new functionality by creating appropriate plug-ins. The Protégé plug-in library contains plug-ins for graphical visualization of knowledge bases, inference-engine for verification of constraints in first-order logic, acquisition of information from remote sources such as UMLS and WordNet, semi-automatic ontology merging, etc. It also provides translators to FLogic, OIL, Ontolingua and RDF(S), and can store ontologies in any JDBC-compatible relational database. Plug-ins, applications and ontologies, which have been developed both by the Protégé group and other Protégé users, are available in the Protégé Contributions. It could be arguable that Protégé is a knowledge acquisition tool and not a library system, but all the extensions that the Protégé community have provided transform it into a more sophisticated system.
Management

- **Storage.** The Protégé-2000 architecture allows for “backend” plug-ins that read and write different storage formats (see Figure 3.4.3.5.2). As a default, Protégé-2000 stores its knowledge bases in a special-purpose flat file format. However, it also has two other built-in storage back-end plug-ins. These allow for reading from and writing to RDF files (Resource Description Framework) and for using a Protégé-specific schema to read from and write to a relational database format. Finally the recent OWL plug-in [Knublauch et al., 2004] provides support for storage, access and maintainance of ontologies expressed in OWL.

- **Identification.** A simple naming mechanism to identify ontologies is defined in Protégé.

- **Editing.** Protégé is created to help knowledge engineers during the acquisition process. The user interface provides many facilities for editors and currently it is the reference for many other ontology editing tools. For instance, undo/redo editing functions, automatic consistency checking and classification (subsumption), and a user-friendly interface.

- **Collaboration.** The OWL Plug-in [Knublauch et al., 2004] includes a very simple collaboration feature to include suggestions about new edition actions through “to-do-list”.

- **Import and reuse.** Protégé currently can be used to load, edit and save ontologies in various formats, including F-Logic, OIL, Ontolingua, RDF(S), CLIPS, XML, UML and relational databases. With the OWL plug-in, Protégé

---

offers native support for OWL ontologies. On the other hand, Protégé provides a good support for ontology inclusion. In Protégé it is possible to reuse definitions from a project by including an entire project. However, the implemented inclusion mechanism has some significant limitations like for instance it doesn't allow extension of included entities. So, it is not possible to re-classify or add a slot to a class in the including model. Further, only the outermost model may be changed, thus making the evolution of dependent ontologies impossible [Maedche et al., 2003a].

- **Export.** Protégé includes also tools for exporting ontologies in several formats like F-Logic, OIL, Ontolingua, RDF(S), CLIPS, XML, UML and relational databases.

- **Versioning and Evolution.** Protégé does not support versioning and evolution features. As we pointed out in Section 3.3.2.4, PROMPTDiff is not a real tool for versioning. It only identifies and formally describes differences between two ontologies.

- **Merging.** iPROMPT (see Section 3.3.2.4) is a semi-automatic tool that guides users of Protégé in the process of merging two ontologies. The tool was originally implemented to deal with ontologies represented in OKBC, but recently was extended to support OWL ontologies.

- **Mapping and alignment.** Through iPROMPT and AnchorPROMPT, Protégé provides some basic features for ontology mapping and alignment. Anchor PROMPT does not work independently, but can improve the results of a proper mapping tool.

- **Factoring.** PROMPTFactor is a semi-automatic Protégé plug-in of the PROMPT suite that includes functionality to help users to extract a complete sub-ontology from an ontology.
Accessibility

- **Searching.** Protégé provides a graphical query tool for searching. No query languages are mentioned in the literature.

- **Browsing.** Protégé uses tree visualization extensively for showing the hierarchical structure of an ontology.

- **Reasoning** (inference across multiple ontologies). Several reasoning engines can be plugged in to Protégé 2000, and with the OWL Plug-in, DL reasoners are also available. As illustrated in Figure 3.4.3.5.3, the OWL Plug-in extends the Protégé model and its API with classes to represent the OWL specification. The OWL Plug-in supports OWL DL and significant parts of OWL Full. The OWL Plug-in provides direct access to DL reasoners such as Racer [Haarslev and Moeller, 2003]. The current user interface supports two types of DL reasoning: Consistency checking and classification (subsumption). Other types of reasoning, such as instance checking, are work in progress.
Standardization

- **Language.** The ontologies can be primarily stored in OKBC, RDF(S) and OWL.

- **Upper-level ontologies.** It is possible to import Upper-level ontologies in the system, but Protégé has to be extended to exploit in concrete applications the advantages of these ontologies.

### 3.4.3.6 KAON (Karlsruhe)

The Karlsruhe Ontology (KAON) tool suite has been developed in the context of the KAON Semantic Web infrastructure. To access an ontology, KAON provides an application programming interface in Java called KAON-API. The same interface is implemented for several storage mechanisms. This allows us to use the KAON front-ends (e.g., OntoMat - an authoring and annotation tool, and KAON-CRAWL - an RDF crawler, etc.) with different ontology storage tools. The user can work with ontologies in several representation languages as long as the representation primitives have been defined to be semantically equivalent to the primitives of the KAON language that works in particular with RDF Schema and DAML+OIL ontologies. KAON-API only supports some simple queries for browsing the ontology (e.g., get all classes or get all resources of a class) and does not allow specifying a filter or an expression path within the query. Figure 3.4.3.6.1 provides a schema of the main elements of the KAON architecture.
Management.

- **Storage.** The suite includes three different back-end implementations of the KAON API (the set of interfaces that allow external applications to access ontologies and instances independent of the physical storage mechanism): the Main Memory implementation, the RDF Server as well as the Engineering Server. The last one is a storage mechanism for KAON ontologies, based on relational databases and is responsible for the concurrent access support.

- **Identification.** The back-end includes nameing mechanisms to identify the ontologies that are stored. Also the Engineering server maintains for each user a local “copy” of the ontology (ontologies) that is concurrently being edited and assigns a version number for each successive state of these ontologies (one for each ontology).

- **Editing.** The OI-Modeler graphical editor provides many features to users for editing purposes. Undo/Redo and generation of additional changes for maintaining consistency are some examples.

- **Collaboration.** No collaboration features are really provided.

- **Import and reuse.** DAML+OIL/OWL, RDF, F-Logic, OKBC.

- **Export.** The same that before.

- **Versioning and Evolution.** KAON includes an evolution mechanism that encapsulates certain policies for evolution with respect to the user’s demands. The evolution strategy tries to automate the generation of a set of additional
changes that preserve the consistency of an ontology after performing a certain type of change. On the other hand, no versioning mechanism is supported.

- **Merging.** No Merging tool is available.

- **Mapping and alignment.** MAFRA (MAppling FRAmework for distributed ontologies) [Maedche et al., 2002a] is a framework defined for mapping distributed ontologies on the Semantic Web based on the idea that complex mappings and reasoning about those mappings is the best approach in a decentralized environment like the Web. MAFRA has been implemented as a plug-in of KAON (an Ontology Management tool developed by the University of Karlsruhe, [http://kaon.semanticweb.org](http://kaon.semanticweb.org)) and introduces two interesting new concepts: Semantic Bridges [Maedche et al., 2002a] and service-centric approaches [Silva and Rocha, 2003].

- **Factoring.** In [Volz et al., 2003], the authors mentioned that they implemented within KAON a prototype with the functionality described in the article that also was summarized in section 3.4.1.5. In summary, Volz and colleagues defined an extension of the query language RQL ([Alexaki et al., 2000a], [Alexaki et al., 2000]) to generate views from RDFS ontologies [Volz et al., 2003a]. They distinguish between two types of views. Views on classes applied to concepts (classes) returns only unary predicates, and views on properties can be defined using arbitrary queries which return binary predicates.

### Accessibility

- **Searching.** KAON provides an experimental conceptual query language (KAON Query) that allows easy and efficient locating of elements in KAON OI-Models. KAON queries do not use variables for selecting subsets of the model. Rather, the query consists of concept and property expressions, which can be arbitrarily combined to create more complex expressions. The actual version of the query language is under development and in a very early stage (the authors mentioned explicitly in [Karlsruhe and Karlsruhe, 2004] that “the current support for querying in KAON is rudimentary”). For example, currently it is not possible to store named queries in the model. As future work, the development team of KAON will provide more description logic features in KAON and will integrate queries with deductive rules.

- **Browsing.** Graph visualization is available in the OI-Modeler.

- **Reasoning** (inference across multiple ontologies). Reasoner engines are classified as External Services within KAON architecture. In the context of the Wonderweb project[59] three inference engines where adapted to be used by KAON (the RDF engine of Sesame, the F-Logic engine of Ontobroker and the DL engine FaCT). Based on the work of Raphael Volz, [Volz, 2004], the KAON inference capabilities were extended to exploit reasoning capabilities of deductive databases for a fragment of OWL called DLP (Description Logic Program). This work is currently being extended by Boris Motik (see DIP

[59] [http://wonderweb.semanticweb.org/index.shtml](http://wonderweb.semanticweb.org/index.shtml)
deliverables D1.1 and D1.3) to support a new fragment which is the result of the combination of a fragment of OWL and function-free Horn clauses. The inclusion mechanism of KAON allows re-using ontologies in the definition of new ontologies. The semantics of the inclusion are described in detail in [Maedche et al., 2003]. Through the inclusion mechanism it is possible to offer reasoning across multiple ontologies.

Standardization

- **Language.** RDF(S) is currently supported in KAON, but KAON’s ontology language extends RDF(S) with some proprietary extensions (see [Karlsruhe and Karlsruhe, 2004]. KAON ontologies consist of concepts, properties and instances grouped in reusable units called OI-models (ontology-instance models). The division between concepts and instances is not strict— an entity can be interpreted as a concept, as well as an instance, depending on the point of view of the observer. An OI-model may include other OI-models, thus having immediate access to all definitions from the included model [Karlsruhe and Karlsruhe, 2004].
- **Upper-level ontologies.** No Upper-level ontologies are specifically mentioned in the literature about the tool.

3.5 Distributed Ontologies – Practical Approaches

This section departs a little from the main thrust of the discussion presented so far in this deliverable. At this point we need to reflect a little on the nature of the project proposed by DIP, and ask the question whether the requirements for an Ontology Management System (and in general for an Ontology infrastructure) are too ambitious. In this section we describe a very simple Ontology Distributed System and show that many issues are still unsolved. The result of this analysis raises the following question: if there are open issues in this kind of simple system, will we be able to provide a complete solution in a more complex framework such as that proposed by DIP? The answer to this question has implications not only for this particular workpackage, but also for the DIP project in its entirety.

3.5.1 Existing Practical Distributed Ontologies

There are several examples of ontologies that have become widely accepted and reused for the purpose of distributed data exchange and integration. Very often these ontologies were organically grown and quickly found a large number of creative users, even though for a long time they were not endorsed by any of the popular standards committees. The most common domains of human activity drew many alternative proposals for the specification of a conceptualization of these domains. Two examples of the most often described domains are represented by ontologies describing a person and ontologies describing a document. Many alternative versions of ontologies describing people and documents are found in online ontology libraries such as Protégé.
Ontologies Library\textsuperscript{60} and SchemaWeb\textsuperscript{61}. The reason for the high frequency of ontologies for describing people and documents is likely to be rather simple: the Web is mostly about people and documents published by the people on the Web. Below, we provide typical examples of the person and document ontologies that have gained a high degree of popularity.

\textit{Person ontologies:}

1) VCard\textsuperscript{62} is a schema to specify an electronic business card profile. Factually, vCard is a simple ontology to describe a person with 14 attributes such as Family Name, Given Name, Street Address, Country, etc. The ontology provides a precise way to describe the instance data using RDF, so that the data conforming to this description can be accessed and reused by other applications.

2) FOAF\textsuperscript{63} (Friend Of A Friend) is a schema which is similar to VCard in that FOAF also is a small ontology to describe a person. The FOAF schema provides 12 attribute types, that are similar to the attributes that vCard provides: First Name, Last Name, Email address, etc., and a precise way to describe the instance data using RDF is also provided by the FOAF-project. However, FOAF is more expressive compared to VCard in that it enables you to create links between people. I.e., you can express with FOAF that s/he knows (is a friend of) some specific person. Thus, FOAF allows one to encode and track connections between people, thus providing more opportunities for practical reuse of ontology instance data. In addition to conventional search and retrieval of the ontology instance data, FOAF provides the means to use personal URIs as data to link people’s semantic annotations in a common network. Thus what is of importance is that FOAF is also one of the ways to support cross-metadata referencing on the Semantic Web.

\textit{Document/web publication ontologies:}

1) Dublin Core provides a vocabulary to semantically annotate web resources and documents. The vocabulary consists of 15 attributes to describe a document or a web resource and contains parameters that express the primary characteristics of the documents, e.g., Title, Creator, Subject, Description, Language, etc. The vocabulary (ontology) is propagated by Dublin Core Metadata Initiative\textsuperscript{64}, an organization dedicated to promoting the widespread adoption of interoperable

\begin{itemize}
  \item Protégé Ontologies Library: http://protege.stanford.edu/ontologies/ontologies.html
  \item SchemaWeb: http://www.schemaweb.info
  \item VCard: http://www.w3.org/TR/vcard-rdf
  \item FOAF: http://www.foaf-project.org
  \item Dublin Core: http://dublincore.org
\end{itemize}
metadata standards and developing specialized metadata vocabularies for describing resources. The goal of promoting a widespread adoption of the standard is claimed to be enabling of more intelligent information discovery systems.

2) RSS is variably used as a name by itself and as an acronym for "RDF Site Summary", "Rich Site Summary", or "Really Simple Syndication". The RSS ontology specifies the model, syntax, and syndication feed format and consists of 4 concepts: “channel”, “image”, “item”, “textinput”, each of them having 3 attributes like “title”, “name”, “description”. RSS was developed in early 1999 to populate Netscape's My Netscape portal with external newsfeeds ("channels") and thus pioneered syndication; that is, provision of a channel of information by representing multiple resources in a single document. Since then RSS has taken on a life of its own and now thousands of Web sites use RSS as a "what's new" mechanism to drive traffic their way.

3.5.2 Ontology Promotion and Distribution in Practice

When a new ontology is proposed, it needs to gain recognition from a considerable community to become a really used standard. Typically, for a popular domain several parties propose and push forward their schemas describing the domain for other parties to adopt. So, reaching agreement as to whose schema is to be used as a standard, and who has to adapt to it, is an important issue that requires a solution. Here, we propose a list of criteria for ontologies that contribute to the promotion and distributed character of ontologies and illustrate these criteria with the successfully expanded formats introduced above.

1) Integration in successful tools for Semantic Web engineering

RSS, VCARD formats are included in Jena-2 [Wilkinson et al., 2003]. Jena-2 is a mature API for OWL, RDF, DAML+OIL data and ontologies, and is recognized as one of the best existing Semantic Web tools at the moment according to the Semantic Web tool assessment by SemWebCentral. The fact that Jena-2 is an open source environment contributes to the affordability and thus widespread popularity of Jena-2. Clearly, integration with Jena-2 for RSS and VCARD formats leads to a broader dissemination and usage of these formats.

2) Extensibility by other ontologies

FOAF ontology is extended by the Relationship ontology that allows a more precise specification of the links between people. The Relationship ontology specifies a vocabulary for describing relationships between people, containing around 20 terms such as “friendOf”, “childOf”, “employedBy”, “worksWith”, “hasMet”, etc. Obviously, 65 SemWebCentral’s Semantic Web tool assessment: http://www.semwebcentral.org/assessment

66 Relationship ontology: http://purl.org/vocab/relationship
being extended by a third party is an acknowledgement of usefulness and appropriateness of the ontology and provides more chances of further reuse and extension.

3) Integration in applications and web resources

Serialization of contents specified by RSS uses resources of BBC, CNET News.Com, iTunes, Telegraph (UK), New York Times, Yahoo! News, wired.com (The news is for a general audience, dealing with technology, culture, business, politics) and slashdot.org (technology) news, etc. On the other hand, involvement of a predefined ontology in an application or a web resource is also likely to lead to inclusion of this ontology in software toolkits developed to support this resource.

4) Simplicity

All the ontologies described above (VCard, FOAF, Dublin Core, RSS) are indeed simple; each of them consists of ca. 15 commonly-known items in a flat structure. Creators of some of these simple ontologies have explicitly stated that the lightweight nature of their ontology in their design goal (e.g., RSS ontology developers put lightweight as the first design goal). Simplicity of an ontology makes it easy for people to understand and simplifies its implementation support and reuse. Its simplicity also contributes to its being multipurpose; thus the same ontology can be reused in different contexts.

5) Based on widely accepted formats

All the ontologies described above (VCard, FOAF, Dublin Core, RSS) have an XML/RDF encoding specification. Since HTML/XML/RDF standards are the main machine-processable formats supported and used on the Web, an ontology proposal has to support these standards in order to make it easy to reuse. Being based on widely accepted standards is also beneficial for interoperability, versioning, and mediation support—third-party tools can be used for the promoted ontologies. Another important issue is extendibility of the XML/RDF standards that facilitates the extendibility of the ontologies designed on the basis of these standards and complies with the trends and objectives of the Web.

3.5.3 Simple Ontologies are not Enough

The RSS working group states that as RSS continues to be re-purposed, aggregated, and categorized, the need for an enhanced metadata framework grows. Channel- and item-level title and description elements are being overloaded with metadata and HTML. Some producers are even resorting to inserting unofficial ad hoc elements (e.g., <category>, <date>, <author>) in an attempt to augment the sparse metadata facilities of RSS.

Many communities who appreciate the usefulness and value of RSS also report that it has reached its limits. There is a demand for more advanced portal syndication which
RSS can not satisfy. One initiative in developing technologies to overcome the limitations of simple ontologies for web publishing comes from Apache Software Foundation and proposes portal syndication with web services and Cocoon (Ivanov, 2004). Another initiative is Atom\(^67\) that is aimed at defining a feed format for representing and a protocol for editing Web resources such as Weblogs, online journals, Wikis, and similar content. The feed format is to enable syndication, and the editing protocol is to enable agents to interact with resources by nominating a way of using existing Web standards in a pattern. To overcome the limits of externally distributed small-scale ontologies, organization of user-driven ontology extension, support and metadata communication within web portals is considered in the approach of the People’s portal [Zhdanova, 2004].

The reason why staying within the scope of simple ontologies (e.g., exchanging FOAF profiles and posting cross-linked news stories from RSS) is not enough and far too limited for the existing Web are as follows:

- embedding and personalizing rich content and behaviour from remote web applications are becoming a necessity for catering to specific user needs;
- extension of simple ontologies, and discovery and communication of these extensions are becoming a necessity for bringing semantics to a larger amount of Web content;
- mapping between simple ontologies and their alignment with other extendible ontologies are becoming a necessity for large-scale data integration.

The solutions introduced by the RSS working group to handle the RSS limitations are as follows: One proposed solution is the addition of more simple elements to the RSS core. This direction, while possibly being the simplest in the short run, sacrifices scalability and requires iterative modifications to the core format, adding requested and removing unused functionality. A second solution, and the one adopted in the RSS specification, is the compartmentalization of specific functionality into the pluggable RSS modules. This is one of the approaches used in this specification: modularization is achieved by using XML Namespaces for partitioning vocabularies. Adding and removing RSS functionality is then just a matter of the inclusion of a particular set of modules best suited to the task at hand. No reworking of the RSS core is necessary.

Obviously, the problems and solutions for RSS ontology above are also valid for other simple ontologies in widespread use. Having simple and easy to understand ontologies and ontology pluggable extensions on the user side, the complex processes of combination and reuse of these ontology components in ever-changing specification and conceptualization processes of the outside world are left encapsulated on the middleware and application side. Clearly, the development and especially reuse of the pluggable extension modules involve complex problems that are not resolved at the moment. These problems arise from the support requirements for practical large-scale extendible ontology management, such as

\(^{67}\)http://www.atomenabled.org
• easy and quick extension opportunity to cater for dynamically arising and changing needs of ontology users;
• discovery of existing pluggable extension modules\(^ {68} \);
• composition of existing pluggable extension modules;
• decomposition of existing pluggable extension modules;
• matching of existing pluggable extension modules and core ontologies with other external ontologies and modules;
• tools to support ontology extensions proposed from the user’s side, discovery, composition, decomposition, matching and reuse of created earlier ontologies and extensions.

Preserving the successful approach of simple usable ontologies and resolution of the issues above are clearly to be considered as major challenges in practical state-of-the-art distributed ontology management.

\(^ {68} \) This requires proper means and methodology for description of the extension. For large multi-purpose ontologies, the problem of context modelling becomes an issue (and it is a well-known tough AI and cognitive science problem).
4 REQUIREMENTS ANALYSIS FOR ONTOLOGY MANAGEMENT IN DIP

On basis of the understanding of Ontology Management in DIP WP 2 as defined in Section 2, and with regard to the re-usability of existing approaches and tools as investigated in Section 3 of this deliverable, the following depicts the requirements for the Ontology Management facilities for DIP to be developed in the course of WP 2.

We discuss the requirements of the DIP Ontology Management Environment from two perspectives (i) the overall project point of view and (ii) those aspects of the development of Ontology Management which may be regarded as the workload for WP 2.

4.1 Requirements Coming from Different Ontology Languages

From the analysis of the different ontology languages a number of conclusions for the DIP ontology management system can be drawn as follows:

- Only one ontology/schema language is not enough, and several communities are using different languages for their purposes. E.g., UML and MOF are used by the software engineering community to model their domain, whereas Topic-Maps are used for document management and browsing, and OWL is used in the Semantic Web world. Focusing on one approach leaves out the others, and therefore instead of unifying the different approaches it is more desirable to provide a common infrastructure which is reusable among all the different approaches. The underlying infrastructure for the ontology management system should be shared among all the different ontology languages. If for every language a different environment needs to be created the overall effort of integrating is too high. Sharing of the overall infrastructure also means sharing of the query language and of the primitives for versioning support, which should be independent of the specific ontology language. As a common data model RDF seems to be a good choice, given that many ontology languages are already based on RDF. Also, since RDF is based on one of the simplest possible data-models, building infrastructure takes considerably less effort than with more sophisticated data-models.

- The ontology management system should also take the semantics of the respective languages into account— e.g., queries asking for all subclasses of a given class should take the transitivity of the subclass relationship into account. Initial ideas on how this can be achieved are delivered by the TRIPLE system [Sintek and Decker, 2001].

- The versioning system needs primitives for versioning as well as collaboration— similar to the CVS system used for distributed software development.

- A pure data-model storage (e.g., RDF storage) is not enough— a versioning system also needs to take different contexts of the sub-graphs into account— e.g., where the data comes from, how it was derived etc.
4.2 Requirements for Ontology Storage and Retrieval

On the basis of our state-of-the-art analysis we can derive a number of requirements for ontology storage and retrieval in DIP. Ideally, a storage and retrieval system would combine the best features of the systems reviewed: the functionality of Jena and Sesame, the performance of relational database systems, the versioning support of Subversion, and the distributed architecture of RDFPeers.

- An ontology storage and retrieval system in DIP must deal with very large ontologies, and must do so efficiently. The ontology store must be tightly connected to higher layers to achieve the performance needed. A context mechanism should be included in the storage scheme to determine the provenance of a certain piece of the ontology. Provenance tracking is especially important if data is integrated from a broad number of sources to help to assess the trustworthiness of statements. Also, the ontology store should support different ontology languages and semantics. As we have shown in 3.1 and 3.2, support of different semantics can be achieved using a semi-structured data model; hence, the ontology storage and retrieval system in DIP should be based on a semi-structured/graph-based data model.

- The query language must support a broad range of features. Conjunctive queries must be supported, data-types such as date and time should be supported, and aggregation functions such as sum() should be available. Range queries and keyword search on literals should be possible and optimized for performance. Optimized operations for ordering the results and fetching results sequentially are a very important feature for using ontology repositories for constructing results pages similar to those of search engines. The query language must provide closure; that means that results of a query can be expressed in the same data model as the ontology, and the query language itself should be encoded in the same semi-structured format as the ontology language to allow for easy processing and storage of queries.

- The access interface must be well defined and should adhere to a standard. Using one standard for accessing all DIP repositories facilitates data integration tasks considerably. We envisagethat it is possible to provide access to all DIP repositories using the virtual integration approach as described in [Harth, 2004]. Furthermore, to allow for easy integration of multiple ontology repositories in a single logical DIP repository, the contents of the repository have to be described using some metadata to enable optimized operations for distributed querying. Also, the repository will be accessed in parallel and therefore must work in a multi-user environment; therefore, the repository must adhere to the ACID paradigm and must include transactions, logging and locking.

- Advanced features such as versioning and fine-grained access control should be supported efficiently. None of the reviewed repositories support versioning sufficiently. The history of an ontology should be stored in a flexible and quickly accessible way. This can be achieved either by storing version information in the indices, or include journals, which track all changes. What type of implementation to choose depends on how often version information is accessed.
We do not have a particular preference for any of the presented repositories. It is very likely that in DIP there will be a large number of different repositories and technologies used by the numerous partners. Thus, it is not very likely that all partners will agree on just one repository. However, it should not matter what repository is used (at least not from a high-level perspective if you want to integrate various repositories) when certain interfaces are implemented. The interface that will be defined by W3C’s Data Access Working Group (DAWG) should be considered as a basis for the access protocol for DIP repositories, possibly with extensions for advanced ontology management functionality, such as support for versioning.

4.3 Change Management

Section 3.3.1.5 summarized the requirements for ontology versioning and evolution identified in [Klein, 2004] and [Stojanovic et al., 2002]. These requisites were the base for the definition of the evaluation schema for the tools analyzed in Section 3.3.2. In our opinion, they should be also the requirements for ontology versioning and evolution in the DIP project. When we were close to finishing this report, Ljiljana Stojanovic (FZI Karlsruhe) sent us her PhD thesis ([Stojanovic, 2004]) where she identified new requirements for ontology evolution. Some of them are included in this section. The requirements proposed for ontology versioning and evolution in the context of the DIP project are described in the following list:

- **Version identification**
  The versioning mechanism of the DIP’s ontology management environment should provide a unique identification for any of the versions of an ontology stored in the system.

- **Version storage**
  The versioning tool should be able to store any of the versions of an ontology in separate locations and to organize the versions in a rational way that simplifies the location of them.

- **Data Accessibility**
  In this versioning requirement, the system should maintain access to instance data from different versions of the ontology, following the two possible strategies proposed by Klein in his PhD thesis [Klein, 2004] and described in Section 3.3.1.5.

- **Formal Change Representation**
  A formal representation of the changes between versions of an ontology is required. In a distributed environment like DIP proposes, the approach of Klein that identifies a (minimal) transformation makes more sense because it would hardly be difficult to main a consistent distributed log of changes. The approach of Stojanovic and colleagues is more feasible for central repositories where a replication configuration is the only distributed schema considered. Nevertheless the proposal of Klein demands more computer resources, so a mixed approach should be considered where a log of changes is adopted to identify changes in distributed ontologies (that are configured in a replication model).

- **Consistent Reasoning**
The versioning module should provide control mechanisms that verify that the modifications added to a new version of the ontology do not generate inconsistencies with the previous ones. The goal of an ontology is thus to evolve an ontology from one consistent stage to the next.

- **Data Transformation**
  The mechanism that provides versioning capabilities should be able to maintain consistency between the instances of an ontology that was modified.

- **Security Support**
  We recommend that the versioning and evolution tool support verification and authorization of the changes that are applied to an ontology.

- **Definition of Evolution Strategies**
  The evolution module should allow users to configure the behaviour of the ontology management system when one of its ontologies is modified. This behaviour is specified using certain policies, and the outcome is the automated generation of a set of additional changes that preserve the consistency of an ontology after performing a certain type of change.

- **Supervision**
  The evaluation of the changes that the ontology management system performs to detect potential inconsistencies, plus the automatic proposition of changes that restore the consistency, are the required capabilities grouped in this section.

- **Transparency**
  The ontology system should provide a visualization mechanism that makes clear to the users the consequences of the application of the changes for the new ontology version.

- **Reversibility**
  The possibility of reversing the effects of the changes that were performed, and to return to a previous consistent state, is another desirable requirement.

### 4.4 Requirements Ontology Library – Searching, Editing and Browsing

The evaluation of Ontology Library Systems in Section 3.4 and the extended analysis of several desirable features in the context of the DIP project (see Sections 3.4.1.1 – 3.4.1.5) allow us to identify a set of requirements that an ideal Ontology Library System for this project should fulfill.

**Management Requirements.** Group all the requirements related with the storage and maintenance of ontologies in a distributed environment. Define security control policies to control access to the information, mechanisms to maintain the coherence of the information distributed or replicated in dispersed locations, re-use ontologies facilities, and concurrent access and collaboration services. These are the main requirements identified. In the following schema, these elements are described in more detail:

- **Storage**
Storage and Retrieval was extensively analyzed in Section 3.2 and a set of requirements were identified in Section 4.2. In summary, we agreed that multi-tier architectures are advisable for a Distributed Ontology Management tool. Modern systems (like KAON or Sesame) use Database Management System (DBMS) as a persistent storage. Several DBMS provide a scalable platform which has many interesting features like access control mechanisms, transaction management facilities or replication support.

- Identification
  An Ontology Library Systems should provide a mechanism to define a unique identifier for each ontology that is stored in the system. The common mechanism is to provide a unique name that identifies each ontology. In a distributed environment it may be advisable to complement this mechanism with the use of URIs as the DAML Ontology Library System does.

- Editing
  Ontology Library Systems in DIP should guarantee concurrent access for editing of multiple users like for example with the definition of locking policies that prevent users from editing the same terms at the same time. Mechanisms to maintain the coherence of several versions of an ontology geographically dispersed should be provided. The execution of remote edition tasks and the concurrent edition of the same ontology are features that are required for an Ontology Library System. Another important element is the visualization facilities that allow users to have a better understanding of the ontology. The solution of OntoRama that combines a Hyperbolic-Tree with Tree visualization is one of the approaches that we prefer. A collaboration infrastructure and evolution techniques for editing support are two other highly desirable elements to overcome the difficulty of editing large and complex distributed ontologies.

- Collaboration
  A collaborative tool is required to simplify the task of concurrent editing of multiple users in a distributed environment. We identified in our analysis that support for concurrent editing is not the same as support for collaboration. Collaboration facilities should set up a virtual environment where users can share ideas, discuss approaches or warn users about certain issues during the editing process. A good starting is Tadzebao, the collaboration tool developed within WebOnto by KMi-Open University.

- Import and reuse
  The possibility of reusing ontologies through references mechanisms like KAON, SHOE or Ontolingua proposed is an important requirement in Ontology Library Systems. Since nobody expects that the ontology will be storaged in a single repository where only one knowledge representation formalism is considered, another important requirement is to provide mechanisms to import ontologies represented in some of the most common ontology languages (OWL or F-Logic for instance) or to provide direct access to heterogeneous sources of information. [Kiryakov et al., 2004] mentioned that they intend to work in this direction.

- Export
In the same way that it is desirable to have mechanisms to import ontologies, it is also necessary to be able to export ontologies using different ontology languages.

- **Versioning and Evolution**
  Requirements for Versioning and Evolution were addressed in a previous section. Here, we want to highlight that identification and storage of the different versions of each ontology, detection and formalization of the differences between versions, accessibility to the instances from the different versions of an ontology and the specification of evolution policies to facilitate the edition of an ontology are the key requirements identified.

- **Merging**
  Merging tools are required because they provide a very powerful mechanism for re-using ontologies through creating a new ontology from two or more existing ontologies with overlapping parts. In our analysis of Ontology Library Systems only Protégé provides such a tool, iPROMPT, a component of the suite PROMPT for Merging, Alignment, Versioning and Factoring.

- **Mapping and Alignment**
  Like the previous one, these techniques are very useful for re-using purposes, and particularly they provide a method to overcome the limitation that it is not always possible to import and merge ontologies. Desired requirements are already identified in [de Bruijn et al., 2004]: Mapping identification and specification, Instance transformation, Instance unification and Query rewriting. MAFRA, the mapping tool implemented within KAON, is an example to follow.

- **Factoring**
  Ontology Library Systems should include factoring mechanisms that allow users to extract complete sub-ontologies. Protégé and KAON are the only tools of the evaluation that implement this functionality\(^{69}\).

**Accessibility Requirements.** These comprise requirements for searching, browsing ontologies and in general for accessing the information that it is stored in Ontology library systems. A desirable element is a reasoning mechanism that allows users of the systems to infer new information for an ontology, or a group of ontologies.

- **Searching**
  Query Languages, Graphical Query Tools and Query Store facilities should be present in the searching mechanism of Ontology Library Systems. Retrieval facilities were studied in detail in Section 3.2 and the requirements were identified in Section 4.2. So, we will focus here on the usability aspect of the Searching mechanisms. We identified three ways to search for information: navigate in the ontology using a visualization ontology tool (see Section 3.4.1.1); query the system using query languages; and query the system using a graphical query tool. The first one is intimately related with Browsing, so it will be analyzed later. Query languages are a very flexible way to search for

\(^{69}\) The plug-in of Protégé PROMPTFactor is the mechanism that provides this functionality.
information and the queries can be re-used in future ones. Finally, Graphical Query Tools are in general very intuitive and easy to use for non-expert users.

• Browsing
Several methods have been analyzed for visualizing ontologies, but not for displaying query results. Since the graphical tools in general increase the usability of the system, a combination of hyperbolic trees with tree visualization (like in OntoRama) should be considered.

• Reasoning
As mentioned in the Section 4.1, the underlying infrastructure for the ontology management system should be shared among all the different ontology languages. RDF is proposed as a common data model (see also [Kiryakov et al., 2004]) and it is also recommended that the ontology management system should also take the semantics of the most common ontology languages into account. The TRIPLE system [Sintek and Decker, 2001], an RDF query, inference, and transformation language for the Semantic Web, is one of the most interesting current efforts in this direction. On the other hand, DIP deliverable 1.1 ([Motik and Spaccapietra, 2004]) defines decidability as a desired requirement for ontology languages that should be complemented with an acceptable performance of the reasoner. The last requirement that will be identified here is the capability to do reasoning through multiple mapped ontologies and through modular ontologies (ontologies that are split and stored in different locations, see [Klein, 2004] and [Stuckenschmidt et al., 2004a]).

Standardization Requirements. Information interchange is one of the core functionalities aimed at in Semantic Web technologies. To achieve this, the DIP Ontology Management Environment has to support existing and emerging standards in ontology languages in order to allow information interchange and interoperability with other Semantic Web enabled applications. The following two are the main requirements that were identified:

• Language
Library Systems should support formal languages for describing ontologies, and in particularly languages that have a broad support like for example OKBC, RDF (S) or OWL. Another possibility is to specify a Neutral Ontology Language API and define the appropriate translators (see [Kiryakov et Al., 2004]).

• Upper-level ontologies
The ontology library system should be ‘grounded’ in some existing upper-level ontologies, such as Upper Cyc Ontology, SENSUS, MikroKosmos, the PENNMAN Upper Model, and IEEE upper-layer ontology.

5 CONCLUSION
Ontology and metadata management is a central problem in realizing the Semantic Web and Semantic Web Services. Collaborative development of shared (possibly heterogeneous) Ontologies requires new data management solutions. Different ontology languages have naturally different requirements for ontology management, ranging from the different kind of necessary inferences dependent on logic the language is based on to how much support it has from the user community. In this report we developed an
evaluation schema which helps us to determine requirements posed by the different ontology languages for ontology management systems.

We examined a variety of different languages, including UML, OWL, RDF Schema, Topic Maps and F-Logic. The languages vary in formalization as well as community-support/usage and expressive power. An ontology management system needs to take the heterogeneity imposed by the different ontology languages and user communities into account and develop a layering of the infrastructure which allows supporting management of Ontologies and metadata for all of the different approaches.

The layering suggested for the Semantic Web also gives hints on how an ontology management is able to deal with the heterogeneity: the semi-structured data standard RDF allows capturing of all kinds of ontology languages (like for example RDF Schema and OWL). An ontology management system layered on top of an architecture based on RDF thus allows reusing as much infrastructure as possible— not only for performing tasks for RDF (like storing and retrieval), but also for performing tasks on the ontology level.

As a consequence, interoperability layers have to be carefully designed to support reusability among different ontology languages as much as possible. One of these interoperability layers is a version support, starting from versioning of the underlying RDF data as a reusable infrastructure component to versioning of the Ontologies. Another necessary component is that the infrastructure component be able to reason with different ontology languages (like UML, Topic Maps, OWL and F-logic), which should be building on top of the basic RDF storage layer. A system with the described layers results in a reusable and widely deployable ontology management system.

Very large ontologies need to be stored in repositories, and the main purpose of a repository is to provide efficient access. In this report we created an evaluation scheme which highlights the various properties of ontology repositories. The premier data management solutions, namely relational databases, are suboptimal for storing ontologies, because they focus on storing structured data. Ontologies are by definition distributed, and if a large number of people work on the same thing you get inconsistencies. A challenge with storing ontologies in databases is that databases require a fixed schema, which is something that cannot be enforced if a large number of people contribute to the ontology in a distributed way. Therefore, an ontology storage system must ensure that the ontology is accessible, even while parts of the ontology can be broken or dysfunctional. Further, the landscape of ontologies in DIP will likely be heterogeneous, since multiple partners may have different requirements for their ontology storage systems.

We showed that semi structured data enables storage and encoding of ontologies with a graph-based model. Various methods exist for mapping that graph-based model into the relational model. Using relational database systems for storing a simple data structure such as a graph incurs a lot of unnecessary overhead. Operations such as versioning, access control and data mining can be very costly, and optimizations for these operations on a database level are difficult and sometimes obscured by the database management system. Furthermore, since low-level operations on repositories (such as retrieve) take place very often, performance problems are elevated to higher layers and multiplied there. A common interface for supported ontology storage and retrieval
solutions would allow partners to use their own storage system while being interoperable.

Ontology stores must provide storage and retrieval operations while allowing access to previous versions of the data and ensure fine-grained access control. Ontology stores must provide these operations efficiently. None of the current implementations satisfy all of the requirements mentioned above. Current repositories are not optimized for performance and lack versioning capabilities and access control. However they support basic storage and retrieval functionalities that can be used right now. When using a defined interface to access repositories, more efficient repositories with extended functionality can be plugged in later for replacement.

Versioning and Evolution is another key issue in the development of a Distributed Ontology Management Server. In evolution, the question is how to ensure consistency of the changed ontology; versioning, in contrast, focuses on how to produce a new version, and how to detect difference between versions. However, the distinction between the concepts of versioning and evolution is far from clear. Klein, [Klein, 2004] claims that there is no difference between versioning and evolution. He merges these two concepts into a new one that it is called “change management”.

It is clear from the discussion in Section 3.3.1.1 that there is as yet no widespread mutual agreement within the scientific community on what the terms Versioning, Evolution and Mediation represent in the field of Ontology. There are important divergences about the ideas presented by Klein and colleagues and Stojanovic and colleagues (two main references in the field of ontology versioning and evolution), but also some agreement on other key issues (like for example the necessity to formalize the specification of changes).

The final element of our study is the evaluation of several ontology library systems and the identification and analysis of a group of relevant features in the context of the DIP project. Visualization and browsing of ontologies, concurrent access and collaboration support are the features that were investigated in detail. The conclusion of our evaluation is that there are no ontology library systems on the market that fulfill with all the requirements that were identified. KAON and Protégé are the only systems that come close to our requirements.

A point of reflection in this deliverable (also extended to the whole work-package) is introduced in Section 3.5. We want to point out that perhaps the requirements for an Ontology Management System in DIP (and in general for an Ontology infrastructure) are too ambitious. Section 3.5 presented a simple Ontology Distributed System and stressed the idea that many issues are still unresolved. So, if there are open issues in the kinds of simple systems outlined in that section, will we be able to provide a complete solution in a more complex framework such as DIP proposed?
REFERENCES


of the 12th International Workshop on Knowledge Acquisition, Modeling and Management (KAWS’99), Banff, Canada, October, 1999.


Sources in TSIMMIS. In: Proceedings of the AAAI Symposium on Information Gathering, pp. 61-64, Stanford, California, March.


