NeOn: Lifecycle Support for Networked Ontologies

Integrated Project (IST-2005-027595)

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

D6.2.1 Specification of NeOn reference architecture and NeOn APIs

Deliverable Co-ordinator: **Walter Waterfeld**

Deliverable Co-ordinating Institution: **Software AG (SAG)**

Other Authors: **Moritz Weiten (ontoprise)**
Peter Haase (University Karlsruhe)

Other Contributors: **Hamish Cunningham (University Sheffield)**
Martin Dzbor (Open University)
Raul de Palma, Óscar Muñoz García (UPM)

Document Identifier:	NEON/2007/D6.2.1/v1.0	Date due:	February 28th, 2007
Class Deliverable:	NEON EU-IST-2005-027595	Submission date:	March 30th, 2007
Project start date:	March 1, 2006	Version:	v1.0
Project duration:	4 years	State:	Final
		Distribution:	Public

2006–2007 © Copyright lies with the respective authors and their institutions.

NeOn Consortium

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

Open University (OU) – Coordinator Knowledge Media Institute – KMi Berrill Building, Walton Hall Milton Keynes, MK7 6AA United Kingdom Contact person: Martin Dzbor, Enrico Motta E-mail address: {m.dzbor, e.motta} @open.ac.uk	**Universität Karlsruhe – TH (UKARL)** Institut für Angewandte Informatik und Formale Beschreibungsverfahren – AIFB Englerstrasse 28 D-76128 Karlsruhe, Germany Contact person: Peter Haase E-mail address: pha@aifb.uni-karlsruhe.de
Universidad Politécnica de Madrid (UPM) Campus de Montegancedo 28660 Boadilla del Monte Spain Contact person: Asunción Gómez Pérez E-mail address: asun@fi.upm.es	**Software AG (SAG)** Uhlandstrasse 12 64297 Darmstadt Germany Contact person: Walter Waterfeld E-mail address: walter.waterfeld@softwareag.com
Intelligent Software Components S.A. (ISOCO) Calle de Pedro de Valdivia 10 28006 Madrid Spain Contact person: Richard Benjamins E-mail address: rbenjamins@isoco.com	**Institut 'Jožef Stefan' (JSI)** Jamova 39 SI-1000 Ljubljana Slovenia Contact person: Marko Grobelnik E-mail address: marko.grobelnik@ijs.si
Institut National de Recherche en Informatique et en Automatique (INRIA) ZIRST – 655 avenue de l'Europe Montbonnot Saint Martin 38334 Saint-Ismier France Contact person: Jérôme Euzenat E-mail address: jerome.euzenat@inrialpes.fr	**University of Sheffield (USFD)** Dept. of Computer Science Regent Court 211 Portobello street S14DP Sheffield United Kingdom Contact person: Hamish Cunningham E-mail address: hamish@dcs.shef.ac.uk
Universität Koblenz-Landau (UKO-LD) Universitätsstrasse 1 56070 Koblenz Germany Contact person: Steffen Staab E-mail address: staab@uni-koblenz.de	**Consiglio Nazionale delle Ricerche (CNR)** Institute of cognitive sciences and technologies Via S. Martino della Battaglia, 44 - 00185 Roma-Lazio, Italy Contact person: Aldo Gangemi E-mail address: aldo.gangemi@istc.cnr.it
Ontoprise GmbH. (ONTO) Amalienbadstr. 36 (Raumfabrik 29) 76227 Karlsruhe Germany Contact person: Jürgen Angele E-mail address: angele@ontoprise.de	**Asociación Española de Comercio Electrónico (AECE)** C/Alcalde Barnils, Avenida Diagonal 437 08036 Barcelona Spain Contact person: Jose Luis Zimmerman E-mail address: jlzimmerman@fecemd.org
Food and Agriculture Organization of the United Nations (FAO) Viale delle Terme di Caracalla 1 00100 Rome, Italy Contact person: Marta Iglesias E-mail address: marta.iglesias@fao.org	**Atos Origin S.A. (ATOS)** Calle de Albarracín, 25 28037 Madrid Spain Contact person: Tomás Pariente Lobo E-mail address: tomas.parientelobo@atosorigin.com

Work package participants

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

Software AG

Ontoprise GmbH

Universität Karlsruhe – TH

University of Sheffield

Universidad Politécnica de Madrid

Open University

Executive Summary

This deliverable describes the NeOn reference architecture, which forms the basis for the software deliverables of the NeOn project. For this architecture important APIs are specified.

In the state of the art section we do not aim for a complete description of engineering infrastructures for ontologies. Instead we investigated several technologies, which have the potential to serve as a foundation for the NeOn infrastructure aiming at large-scale, dynamic and networked ontologies.

Based on these investigations and on the requirements gathered in the previous deliverable of this NeOn infrastructure work package a reference architecture is developed. In order to serve as the foundation for the software deliverables of the NeOn project emphasis is put on the extensibility and plugability of the infrastructure. Nevertheless it is essential for the usability and scalability goals of NeOn that the architecture has a clear focus in order to allow a good interaction of tools.

An example for the realisation of that approach is the adoption of the dual language reasoning by supporting the description language OWL/DL and a rule language. This achieves on the one hand a very good coverage of the almost all ontology language features. On the other hand it has a clear focus on the two most relevant and sufficiently distinct ontology languages instead of trying to have native support for an arbitrary number of ontology languages with minimal differences.

For the NeOn reference architecture the important APIs are described. It contains APIs defining lower-level infrastructure functionality. These allow to have different realizations of those components. Additionally it contains APIs which allow to plugin ontology functionality developed in the technical work packages of the NeOn project. Thus a stepwise completion of the NeOn toolkit as the reference implementation for the NeOn architecture is possible.

In order to illustrate the architecture finally example components and their assembly to example configurations are given. The example components represents typically category of components as well as typically ontology engineering functionality developed by the technical work packages of the NeOn project.

2006–2007 © Copyright lies with the respective authors and their institutions.

Table of Contents

NeOn Consortium 2
Work package participants 3
Executive Summary 3
Table of Contents 4
List of figures 7
List of tables 8
1. Introduction and Motivation 9
1.1 Functional Aspects of the NeOn Approach 9
1.2 Methodological Approach 10
1.3 Overall structure of the NeOn platform 10
1.4 Starting Point and Goal 11
2. Definitions, Acronyms and Abbreviations 13
3. State of the Art 14
3.1 Supporting Technologies 14
3.1.1 Eclipse as IDE Technology 14
3.1.1.1 The Plugin Concept *15*
3.1.1.2 IDE Elements: Views, Perspectives, Editors *16*
3.1.1.3 OSGI *17*
3.1.1.4 Eclipse as a Base for IDE Tools *19*
3.1.1.5 Relevance for the NeOn Reference Architecture and the NeOn Toolkit *20*
3.1.2 Other supporting technologies 21
3.1.2.1 SOA infrastructure *21*
3.1.2.2 Service Data Objects (SDO) *21*
3.1.3 Technologies for language processing 22
3.1.3.1 GATE *23*
3.1.3.2 UIMA *26*
3.1.3.3 ISO TC37/4 ..... *29*
3.2 Existing Infrastructures 31
3.2.1 Components of existing Ontology Engineering Tools 31
3.2.1.1 OntoStudio *33*
3.2.1.2 Protégé 3.2 beta *34*
3.2.1.3 Altova SemanticWorks 2006 ..... *37*
3.2.1.4 TopBraid Composer *38*
3.2.1.5 SemTalk 2 ..... *40*
3.2.1.6 Integrated Ontology Development Toolkit (IODT) *40*
3.2.1.7 SWOOP v2.3 beta 3: *42*
3.2.2 Summary and Conclusions 44
4. General Architecture 47
4.1 Core concepts 47

4.1.1 Layering of architectural components 47
4.1.1.1 Functionality vs. non-functional requirements 48
4.1.2 Service interfaces 49
4.1.2.1 Service-oriented vs. Object-oriented 49
4.1.2.2 Transport layer 49
4.2 Development architecture 50
4.2.1 Use Cases 51
4.2.2 Layer 3: GUI components 52
4.2.3 Layer 2: Engineering components 52
4.2.3.1 Coupling of components 54
4.3 Layer 1: Basic Infrastructure Services 57
4.3.1 NeOn Reasoning Service 58
4.3.2 Ontology Repository 58
4.3.2.1 Core Functionality of repository 58
4.3.2.2 Optional repository functionality 59
4.3.2.3 Realisations 60
4.3.2.4 File based 60
4.3.2.5 RDBMS based Repository 60
4.3.2.6 RDF-based 61
4.3.2.7 XML-DBMS based Repository 61
4.3.2.8 Datastore 61
4.3.3 Ontology Registry 62
4.3.3.1 OMV specialized ebXML registry 63
5. Language API 67
5.1 Role of the API 68
5.2 API 69
6. Example Configurations of NeOn Architecture 77
6.1 Information Integrator development environment as example configuration 77
6.1.1 Ontology generation 77
6.1.2 Mappings / Networking 78
6.1.3 Query development 79
6.1.4 Deployment 80
6.1.5 Querying data sources 80
6.2 Example Engineering components 81
6.2.1 Tightly coupled component Term Translator (UPM) 81
6.2.2 Interactive component Ontology Navigation (OU) 86
6.2.3 GATE as loosely coupled component (USheffield) 87
6.2.3.1 GATEService (aka GaS) 87
6.2.3.1.1 GATE mode 88
6.2.3.1.2 Remote DataStore 88
6.2.3.2 Web-Service interface 89
6.2.4 NeOn OWL Editor Prototype as example for NeOn Ontology meta model based generation (UKarl) 90
6.2.5 FOAM (UKarl) 90
6.2.6 ORAKEL (UKarl) 90
7. Mapping to Requirements 93

2006–2007 © Copyright lies with the respective authors and their institutions.

7.1 Features ... 93
7.1.1 Networked Ontologies ... 93
7.1.2 Registry ... 94
7.1.3 Repository ... 94
7.1.4 Language-Model ... 94
7.1.5 Inference Machine ... 94
7.1.6 Runtime support ... 95
7.1.7 Integrated Development Environment (IDE) ... 95
7.1.8 Ontology Discovery and Analysis ... 95
7.1.9 Usability ... 95
7.1.10 Context and customization ... 95
7.1.11 Lifecycle support and Collaboration ... 96
7.1.12 Heterogeneity ... 96
7.1.13 Modularity/Extensibility ... 96
7.1.14 Integration with the "Non-Semantic World" ... 96
7.1.15 Unstructured Information Management ... 96
7.1.16 Multilinguality ... 96
7.1.17 Security ... 97
7.1.18 Web Integration ... 97
7.1.19 Multiplatform ... 97
7.1.20 Open Source ... 97
7.1.21 Documented ... 97

8. Trademarks ... 98

9. References ... 99

List of figures

Figure 1: “Pillars” of the NeOn approach 11
Figure 2: Eclipse Core Components 14
Figure 3: The Plugin concept of Eclipse 15
Figure 4: Elements of the Eclipse IDE – Java Editor as example 16
Figure 5: OSGI Architecture 17
Figure 6: SDO components 22
Figure 7: Architecture of the Protégé OWL 35
Figure 8: Protégé 3.2 Beta 37
Figure 9: Altova SemanticWorks™ 38
Figure 10: TopBraid Composer 39
Figure 11: SemTalk 40
Figure 12: Architecture of IODT [LiMa2006] 41
Figure 13: IODT graphical editor 42
Figure 14: Swoop Architecture 42
Figure 15: SWOOP GUI 44
Figure 16: NeOn Toolkit Architecture 47
Figure 17: Core Components 50
Figure 18: Ontology Engineering Use-Cases (Examples) 51
Figure 19: Extension Use-Cases (Examples) 52
Figure 20: Plugin-Categorization: Model-Access 53
Figure 21: Schema Editing Plugins 53
Figure 22: Selection Context 54
Figure 23: Loosely coupled engineering components 56
Figure 24: Repository functionality 62
Figure 25: OMV Model 64
Figure 26: API Use-Cases 67
Figure 27: Relation between generated Metamodel implementation and Language API 68
Figure 28: Current Status of Infrastructure 69
Figure 29: Ontology Management Classes 70
Figure 30: Handling of Requests 71
Figure 31: Handling of Queries 71

2006–2007 © Copyright lies with the respective authors and their institutions.

Figure 32: Interfaces of the first-order logic API (part) ... 72
Figure 33: Interfaces of the first-order logic API (part) ... 73
Figure 34: Axioms Interfaces ... 74
Figure 35: OWL Elements Interfaces ... 75
Figure 36: F-Logic Interfaces ... 76
Figure 37: Application landscape ... 77
Figure 38: Generating initial ontologies ... 78
Figure 39: Integrating ontologies ... 79
Figure 40: API structure [Gantner2004] ... 82
Figure 41: Architecture ... 83
Figure 42: Watson on the level of NeOn distributed components/services ... 84
Figure 43: Watson on the level of NeOn toolkit GUIs ... 86
Figure 44: Overview of the ORAKEL system ... 91
Figure 45: Pyramid [Leffingwell2003] ... 93

List of tables

Table 1: OWL Editors: general information ... 45
Table 2: OWL Editors: Interoperability ... 45
Table 3: OWL Editors: Reasoning ... 46
Table 4: OWL Editors: OWL-DL KR primitives ... 46
Table 5: Steps, sources and techniques used for localizing in LabelTranslator ... 81

1. Introduction and Motivation

In the past years semantic technologies have become increasingly popular as one of the important answers to the challenges of information management. A number of approaches and concrete implementations as well as first industrial applications are one consequence of this trend. This includes research prototypes as well as production-stable technology. There is infrastructure on the backend-side (repositories, reasoners, servers) as well as on the front-end side (editors, annotation technology) that can be regarded as stable and mature.

At the same time semantic technologies face major challenges. In the current phase of knowledge system development ontologies are produced in larger numbers and exhibit greater complexity. Applications require a high degree of flexibility and openness of implementations – on the data model level as well as on the technical level. Popular tools available today for ontology development are limited with respect to (i) lifecycle support, (ii) collaborative development of semantic applications, (iii) web integration, and (iv) the cost-effective integration of heterogeneous components in large applications.

All those issues are addressed within the NeOn project and the design of the reference architecture. We look at existing semantic technologies as well as supporting technologies to integrate established approaches that have proven to be efficient and flexible. At the same time we take up new approaches regarding the ontology engineering process, ontology-based applications and related technologies.

A main goal of the NeOn architecture is to establish an infrastructure rather than just an editor. We take into account the network aspect of ontologies, which we address by the integration of registry-technology, a well-established approach for networked resources.

This document mainly covers architectural aspects, the role of pre-existing components and the language API. The seed of the NeOn toolkit implementation is described in project deliverable D6.3.1. The implementation of a sample reasoning service is covered in project deliverable D6.5.1.

1.1 Functional Aspects of the NeOn Approach

Lifecycle Support and Collaboration

Ontologies evolve dynamically rather than in a waterfall-like manner. They are developed, changed and reused. Ontologies are merged or modularized. However, current Ontology Engineering environments lack the required "lifecycle awareness" to support this accordingly.

Heterogeneity

Users looking for an infrastructure covering different paradigms as well as different aspects of semantic applications however still face a heterogeneous set of editors, runtime environments, etc. Whereas standards on the language and interface level lay the ground for interoperability, the technologies have not at all reached their potential as they could be much more than the sum of all parts.

Integrating with the "Non-Semantic World"

The obvious integration with the "non-semantic world" is the transformation of all kinds of meta-data like interface signatures, data base structures and UML models into ontologies. This must be possible in both directions and can be extended for certain use cases also to instance data.

Less obvious is the operational integration of semantic technologies with other technologies. This means that realisations of semantic technology should interact smoothly with typical components of a certain infrastructure. Examples are SOA components, Security infrastructures or Eclipse

2006–2007 © Copyright lies with the respective authors and their institutions.

plugins. This often means that at least at the interface level not all data or its descriptions are realized with ontologies.

Language-Model

NeOn incorporates the notion of networked ontologies. NeOn also has the goal to be compliant with common semantic web paradigms and standards.

Rule Support

For some time now rules are considered an important modelling paradigm and representation form. Standardization efforts and the discussion about the right approach are ongoing and show first results [Ginsberg2006]. Implementations supporting rules (in line with existing languages or proposals for extensions) are available. However, we believe rule-capabilities should not just be “attached” to editing tools. Rule-support should be offered in an integrated way, including debugging and profiling capabilities.

1.2 Methodological Approach

For the development of the NeOn architecture two main sources have been used. First the NeOn architecture requirements and second an analysis of relevant architectural concepts in other engineering platforms. Based on these different variants for all architectural issues have been discussed within participants of the infrastructure work package. This document contains the discussion of the most important issues. This resulted in a consistent NeOn architecture. For verification the NeOn architecture has been applied to different configurations.

Software Architectures

This document describes an architecture for semantic web technology. While “architecture” frequently appears in product- and project descriptions, it’s often used in different ways. The ANSI/IEE definition of “architecture” is given below:

> “Architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution." [IEEE2000]

According to this definition an architecture is a rather high-level concept. It does not directly imply the usage of a certain platform (e.g. programming language, system) or framework (library, etc.). Therefore architecture is not to be mistaken with application design. However, the definition given above (as well as other definitions of the term) still gives much room for interpretations.

The generic nature of the term architecture and the divergence regarding common interpretations makes it necessary to introduce our notion of architecture in the NeOn context.

On one hand we provide the general concepts of the NeOn architecture in line with the definition given above. This is reflected by a layered approach and generic components. On the other hand we take into account concrete ontology engineering functionality.

1.3 Overall structure of the NeOn platform

We describe here the general structure of the NeOn platform.

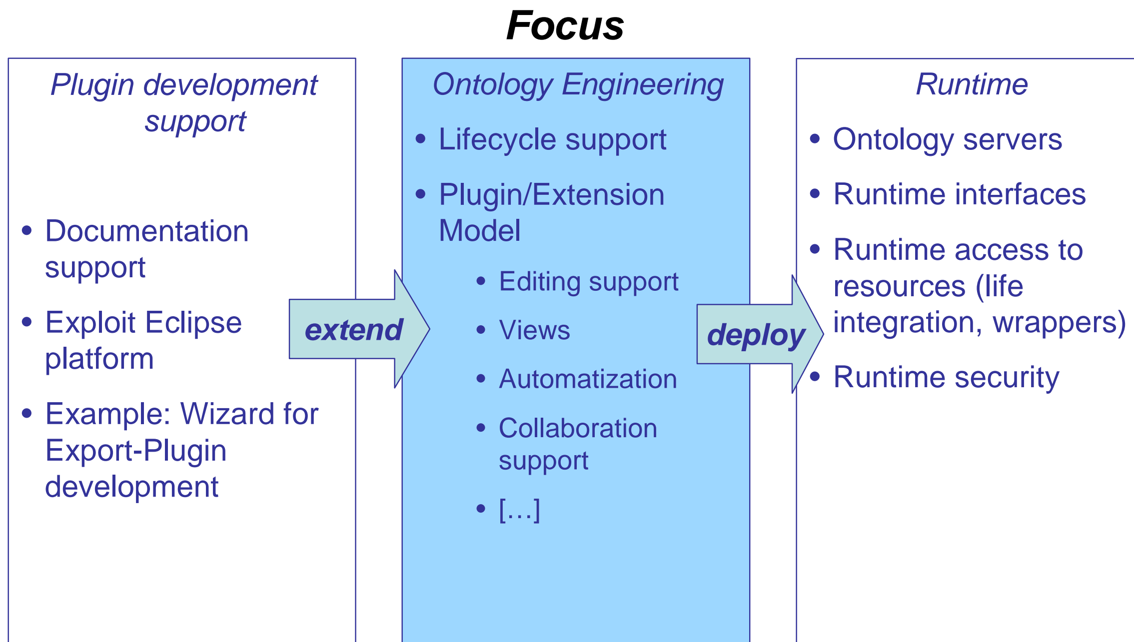

Figure 1: “Pillars” of the NeOn approach

As shown in figure 1, NeOn covers three different levels of a semantic web infrastructure. One is the ontology engineering support comparable to existing ontology editors like Protégé or Swoop. This is clearly a focus of the NeOn activities. As mentioned above we go beyond the “traditional” notion of an editor as we integrate network- and lifecycle aspects.

Applications based on semantic technologies need runtime support. As illustrated by the right column, this is supported by the NeOn approach through a basic runtime infrastructure as well as deployment capabilities.

The infrastructure established by NeOn is meant to be open and extensible. Since NeOn builds on a powerful and at the same time complex platform, development support is a crucial aspect. NeOn will provide core ontology engineering functionality like editing, graphical visualisation and collaboration support. It will also have powerful extensibility mechanisms to add specific components to the toolkit.

1.4 Starting Point and Goal

The NeOn project starts with a number of existing technologies and tools. Those can be compared to existing counterparts e.g. existing ontology engineering environments, reasoners, information extraction tools, etc. The NeOn reference architecture and the NeOn toolkit as an implementation will (i) extend the capabilities of the single tools or components (e.g. provide lifecycle support for ontology engineering) and (ii) provide an infrastructure that goes beyond current environments.

The project portfolio includes (but is not limited to):

- A set of ontology engineering tools available as standalone tools or components (see section 5.2).
- An ontology engineering environment with a focus on rules (OntoStudio, see section 2.3.1.);

- Two mature and fast reasoners/ontology servers (see section 2.3.);
- Registry- as well as repository-technology (CentraSite, Oyster and Watson see section 3.3.2. and 5.2.2.);
- An extensive platform for natural language processing applications (GATE, see section 2.1.3.1.);

All technologies will be enhanced by themselves, e.g. by establishing OWL support for OntoStudio, developing a ebXML-based interface for Oyster and Centrasite and creating a hybrid reasoning platform. However, the main achievement will be the evolution of an infrastructure that

- provides life-cycle support through the coupling of ontology engineering technology and backend infrastructure like repositories and registries;
- is based on a common metamodel and API for both OWL and (frame-based) rules;
- provides multilinguality;
- offers a wide range of engineering functionalities on different levels (graphical, textual, form-based);
- covers ontology engineering, runtime systems and the development support for extensions.

2. Definitions, Acronyms and Abbreviations

F-logic	Frame Logic http://portal.acm.org/citation.cfm?id=210335
FOL	First Order Logic
OSGI	Open Service Gateway Interface An OSGi Service Platform provides a standardized, component-oriented computing environment for cooperating networked services. http://www.osgi.org/
OWL	Web Ontology Language Recommendation by the W3C http://www.w3.org/TR/owl-features/
OWL-DLP	Web Ontology Language - Description Logic Programs, a.k. OWL Easy, Subset of OWL, containing all OWL Syntax could be explained in Horn Logic http://logic.aifb.uni-karlsruhe.de/
RDF(S)	Resource Description Framework Schema http://www.w3.org/RDF/ and http://www.w3.org/TR/rdf-schema/
RIF	Rule Interchange Format (W3C working group) http://www.w3.org/2005/rules/
SDO	Service Data Objects http://www.osoa.org/display/Main/Service+Data+Objects+Home
SPARQL	Query Language for RDF http://www.w3.org/TR/rdf-sparql-query/
SWRL	Semantic Web Rule Language (W3C submission) http://www.w3.org/Submission/SWRL/
WebDAV	HTTP Extensions for Distributed Authoring http://www.ietf.org/rfc/rfc2518.txt

2006–2007 © Copyright lies with the respective authors and their institutions.

3. State of the Art

3.1 Supporting Technologies

3.1.1 Eclipse as IDE Technology

A complete discussion of the Eclipse platform as well as an in-depth comparison to comparable frameworks (like Netbeans) is out of the scope of this document and covered by a number of books and online articles [Clayberg2006]. We provide a basic introduction and highlight the aspects that are especially relevant for the design of the NeOn reference architecture and the NeOn toolkit.

As stated on the website of the eclipse foundation (www.eclipse.org, January 2007) eclipse is

> *"an open source community whose projects are focused on building an open development platform comprised of extensible frameworks, tools and runtimes for building, deploying and managing software across the lifecycle."*

A major outcome of the Eclipse community is the eclipse platform and numerous extensions. The eclipse platform serves as a generic Integrated Development Environment (IDE). The Eclipse platform is highly modular. Very basic aspects are covered by the platform itself, such as the management of modularized applications (plugins), a workbench model and the base for graphical components.

Despite being open and extensible the Eclipse IDE predefines the design of applications (specialized IDEs) that use it. However, since Eclipse version 3 applications that do not follow and IDE paradigm (or one that is different from the Eclipse IDE) can use the non-IDE part of the Eclipse core, named Rich Client Platform.

The Eclipse IDE specifies the structure of the main menus and windows.

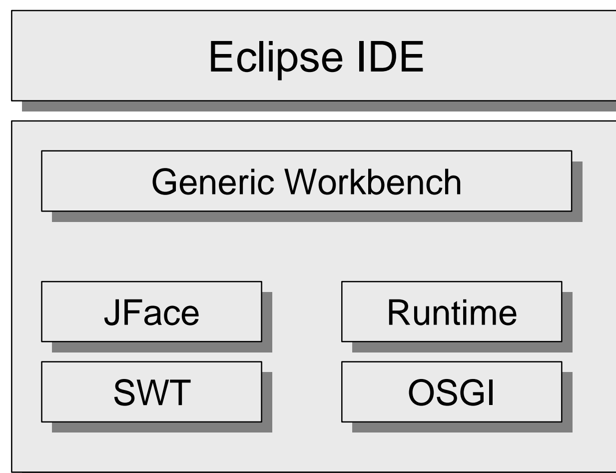

Figure 2: Eclipse Core Components

Figure 2 shows the basic components of the Eclipse platform. This includes libraries for graphical components (SWT/JFace), the platform runtime and a generic workbench. The platform runtime is based on an implementation of the OSGI specification, which we'll discuss in section 3.1.1.3.

3.1.1.1 The Plugin Concept

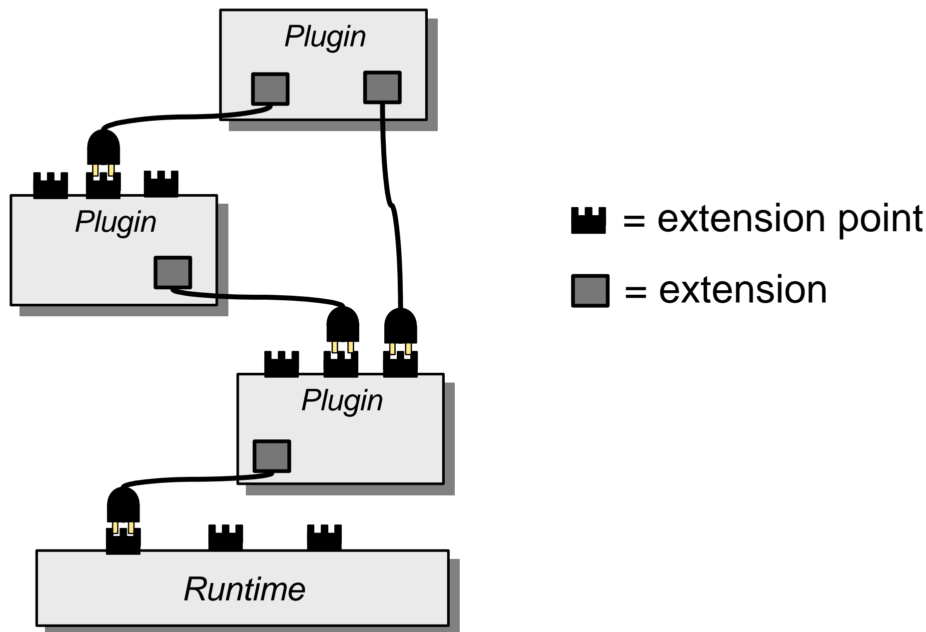

Figure 3: The Plugin concept of Eclipse

Figure 3 shows the plugin concept of Eclipse. Plugins are not limited to certain aspects of the IDE but cover all kinds of functionalities. Eclipse has become popular as a Java-development environment. However, the Java-development support is not provided by the Eclipse platform but by a set of plugins. Even functionalities users would consider to be basic (such as the abstraction and management of resources like files or a help system) are realized through plugins. This stresses the modular character of Eclipse, which follows the philosophy that “everything is a plugin”.

A plugin itself can be extended by other plugins in an organized manner. As shown in figure 3 plugins define extension points that specify the functionality which can be implemented to extend the plugin in a certain way. An extending plugin implements a predefined interface and registers itself via a simple XML-file. In the XML file the kind of extension as well as additional properties (such as menu entries) are declared.

This concept is supported through an extension registry and the (possible) dynamic loading of extending classes. The registry stores the metadata of implementations fitting into a certain extension point. At a certain point the extension point provider triggers the registry to load the extensions that have been registered. This can be done in very dynamic manner, loading the implanting classes just when they are about to be used.

In addition to the extension-mechanism plugins can be connected directly on the API-level (sharing the code). The access to plugin-code can be controlled beyond the level of the Java-language. Packages can be made visible to extending plugins. The base for this is the OSGI implementation of Eclipse.

The Plugin-concept of Eclipse supports flexible architectures based on a common paradigm for the user interface. The existing plugins cover a wide range of functionalities to support typical development tasks and organize resources.

2006–2007 © Copyright lies with the respective authors and their institutions.

3.1.1.2 IDE Elements: Views, Perspectives, Editors

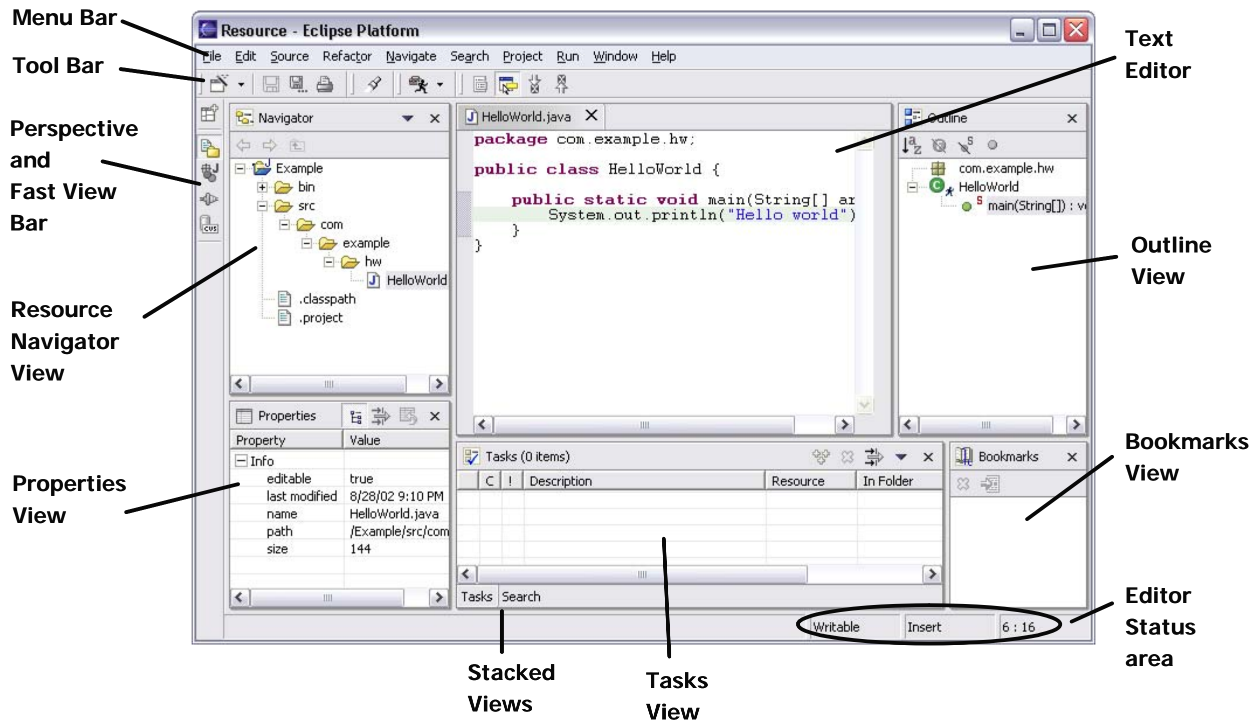

Figure 4: Elements of the Eclipse IDE – Java Editor as example

The main elements of the Eclipse IDE are shown in figure 4 above, using the Java Editor (as part of the JDT plugins) as an example. The basic structure is predefined in a typical IDE design. A main menu bar and a tool bar are the top-level entry points for tasks on the project- or workbench-level, such as the import of resources (files, projects, …).

Views and editors are elements users mainly interact with when performing design- or development tasks. Editors include (but are not limited to) textual editors, such as in the example in figure 4. They implement the open-edit-save-lifecycle for a resource. Views are containers for GUI components such as tree controls and tables. They might just display non-editable information such as the metadata of a file or provide editing or managing functionalities. Views are standardized components. They can be extended in certain ways, e.g. by additional menu items.

A perspective is the definition of the arrangement of views and editors. This includes default settings regarding the size and the position of views as well as restrictions. Perspectives can be adapted by the user. They are often used in a certain context. The same resource might be accessed in different contexts (like a text-file, a repository or a certain element of a datamodel, such as a rule). The context could for example be “editing” or “debugging” (of rules, models, programs, …). Certain editors or views might be useful in both contexts, while others are only used in one specific context. The concept of perspectives helps to organize this in a very flexible manner.

A view for problems or “To-dos” might be present in different contexts. A view showing temporary values of variables (in a rule) as the result of a debug-run would only make sense in the context of debugging.

3.1.1.3 OSGI

OSGI (Open Service Gateway Initiative)[1] was originally developed for embedded systems. OSGI specifies an open, java-based platform for services [OSGI2003]. It goes back to the OSGI alliance, which was founded in 1999. The OSGI alliance provides the following definition:

> The OSGi™ specifications define a *standardized*, *component oriented*, computing environment for *networked services* that is the foundation of an enhanced *service oriented architecture*.
>
> [OSGI2005]

OSGI defines a lifecycle model and a registry for services. Services are specified by Java™-interfaces. It supports server-based administration and provides a number of standard services. Those cover administrative tasks, security, protocols and I/O (and more).

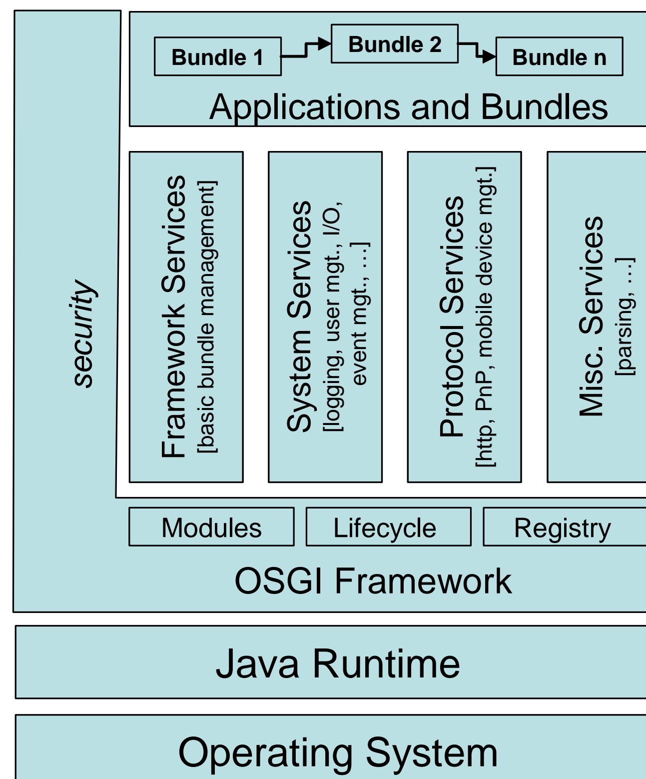

Figure 5: OSGI Architecture

Figure 5 shows the OSGI architecture. The basic building blocks are the OSGI framework and a set of standard services. The framework itself provides modularization of applications or bundles[2] (using class-loading policies), lifecycle management (e.g. starting and stopping applications) and a service registry, that allows the dynamic interaction of services (sharing of objects). Security features are provided orthogonal to those functions ("ubiquitous security"). It uses security

[1] Not to be mistaken for OGSI, the Open Grid Services Infrastructure

[2] Usually modularized applications are referred to as "bundles" in the OSGI context

2006–2007 © Copyright lies with the respective authors and their institutions.

mechanisms of the Java platform (VM security features, language features) and adds security mechanisms for bundles on top of this.

Being designed for embedded devices, OSGI is designed for remote management. OSGI itself does not enforce a protocol. It allows the encapsulation of protocols by API calls, and provides this API through the management system.

OSGI supports the sharing of code as well as services. Code sharing is useful if different (dynamically managed) applications use the same library. OSGI provides means of dealing with dependencies on the same library but in a different version.
Services allow to share objects, which are specified by interfaces. This is supported by a registry, which enables bundles to register objects with the registry, search the Service Registry for matching objects and receive notifications when services become registered or unregistered.

The dynamic support for the modularization of applications is one the characteristics that has made OSGI attractive outside the context of embedded devices. OSGI implementations (see below) cover a range of applications proving this fact.

OSGI Implementations

There are a number of OSGI implementations, including (but not limited to):

- Knopferfish OSGI (see http://www.knopflerfish.org/)
- Apache Felix (see http://cwiki.apache.org/FELIX/index.html)
- Eclipse Equinox (see http://eclipse.org/equinox/)

The last one has gained some popularity due to the fact that it's part of the eclipse platform since release 3.2. It supports OSGI R4, including *declarative services*. The latter are a certain form of dependency injection, which allows the configuration of dependencies and the declaration of services based on XML-files. This includes an "intelligent" class loading strategy, resolving dependencies dynamically ("lazy loading"). However, eclipse 3.2 does not yet support declarative services in such a way that tool support is available in a similar way as for the "traditional" eclipse extension mechanism.

Relation to the Eclipse Plugin Concept

There is a strong relation between the Eclipse Plugin concept and the Eclipse OSGI implementation. This is however not a 1:1 mapping. The OSGI runtime wasn't the base for Eclipse bundles until version 3.0 was released. Since then the plugin-support is based on the OSGI implementation. However, there is a functional overlap between the principle capabilities of OSGI and the realization of the plugin concept on top of the OSGI implementation. The Eclipse plugin concept does currently not exploit the service-based capabilities of OSGI. The latter includes so called *Declarative Services* (since OSGI R4). Declarative services support the management of service dependencies in a dynamic environment. The extension mechanism of Eclipse was originally developed as a static concept. The dynamic management of extensions (adding and removing them at runtime) was introduced on top of the existing framework introducing some additional complexity. OSGI-services have a dynamic character per se. However, the management of service dependencies can imply difficulties for applications using those services. This is the case if the registration process is not centralized but bundles containing services are responsible for the registration. Declarative services use a special component that handles the registration on behalf of them. That component provides proxy objects to service consumers and loads the corresponding bundle when the service is actually invoked.

Despite the fact that the Eclipse OSGI implementation supports declarative services there is currently no similar tool support for extensions. And for desktop applications running on a single

machine (the common IDE scenario) extensions have proven to be sufficient. However, for server-side applications the concept of declarative services has some significant advantages.

OSGI for Server-Side Applications

Recently there have been several approaches to combine OSGI implementations with server-based technology. This is seen as a possibility to combine the advantages of OSGI (modularization, dynamic management of modules, versioning of modules) with the capabilities of application server technology. This can be achieved by either embedding OSGI on an application server (leaving start-up control to that server) or embedding web-server technology in the OSGI runtime. The latter option just means to create a specific web-server bundle and integrate it.

3.1.1.4 Eclipse as a Base for IDE Tools

As we have seen in the previous sections, Eclipse offers a complete framework for IDE applications. Some main advantages and disadvantages regarding Eclipse as the base for IDE applications are summarized below:

- ☺ Eclipse provides an extensible and modular platform.
- ☺ Eclipse provides a common UI paradigm well-known to a large community.
- ☺ Eclipse provides a large set of plugins covering a wide range of functionalities including frameworks for editors.
- ☺ There are Eclipse plugins providing extensive capabilities for meta-model (or model-driven) approaches.
- ☺ Eclipse runs on different platforms.
- ☺ Eclipse is based on an OSGI implementation that can operate in “headless mode” (without GUI) offering options for modularized, dynamic and remotely managed server applications.
- ☺ Eclipse offers a system-specific Look and Feel.
- ☹ The Eclipse platform is rather complex.
- ☹ The default Update-Manager of the IDE is not as comfortable as it could be (commercial alternatives are available).
- ☹ SWT/JFace is outside of the Java-Specification.

Plugin concept and functionalities of existing plugins provide a very efficient base for IDE applications. However the complexity of the platform can be a challenge for users as well as developers. This is addressed by the Eclipse community and plugin-providers through an extensive online help, the concept of “cheat sheets” (interactive tutorials) and numerous articles, tutorials, etc.

For Eclipse-based applications targeting a community of users and developers offering this kind of support is an important aspect. Later in this document we describe how this is addressed within the NeOn toolkit by (i) a categorization of plugins that is defined in one abstraction layer above the Eclipse plugin concept and (ii) the creation of supporting tools. Another important aspect when using not only the Eclipse platform but also existing plugins are the underlying paradigms and how they relate to an IDE for semantic technologies.

2006–2007 © Copyright lies with the respective authors and their institutions.

Paradigms

The overall paradigm of Eclipse is that of an Integrated Development Environment (IDE). While there is no precise definition for an IDE and the attributes associated with that term, it's usually associated with programming languages and the concepts that are common to the most wide-spread IDEs. However, those are designed for common programming languages such as C++ and Java. Some of the principles regarding the development process supported by an IDE are rather generic while others can't be directly applied to semantic technologies. Here are some examples:

- ✓ A development environment needs to provide editing capabilities for defined formats. This typically includes textual editors (e.g. for Java source files) but might also include graphical editors (e.g. in form of UML support).
- ✓ A development environment needs to provide a framework that has a notion of languages with a defined syntax (influencing the principles of editors, such as error handling).
- ✓ A development environment has to offer a concept to organize the visualization and the manipulation of a certain datamodel (e.g. by form-based components, wizards, etc.).
- ✓ A development environment needs different perspectives on resources.
- ✗ In contrast to the semantic of ontology languages most well-known programming languages are procedural languages. On a more fine-grained level this influences the concepts of an IDE. The debugging concept is an example: Eclipse provides a framework for debuggers in form of a set of plugins. It is a language independent framework supporting the development of specific debuggers. However, certain characteristics such as the intended workflow are related to a procedural paradigm. Rules in the context of semantic technologies on the other hand have a declarative semantics. This can imply problems regarding the use of the provided framework.
- ✗ Most common programming languages are consequently (text-)file-based. This means that the development support is largely based on file-handling techniques and typically assumes a permanent synchronization between the text-file and an in-memory representation of the datamodel. The "backend" of the IDE usually is a file-repository and many of the supporting technologies (team support, etc.) assume the existence of files that do not extend beyond a certain size. Ontology languages are not as tightly bound to a file-paradigma. Current editors like Protégé or OntoStudio have an internal repository that does not need files as "backend".

3.1.1.5 Relevance for the NeOn Reference Architecture and the NeOn Toolkit

The aspects listed in the previous section are just some examples showing that the choice of a platform like Eclipse can reduce the amount of work to establish a basic IDE infrastructure significantly and offer additional possibilities through frameworks for certain tasks.

The last two aspects show that (i) complexity has to be targeted explicitly and (ii) concepts of the platform and plugins need to be checked against their applicability in the NeOn context. However, despite this overhead we believe that the choice of Eclipse as a base platform, which is one of the NeOn requirements, is a strategic decision.

There are a number of eclipse-based frameworks that are candidates to support the NeOn infrastructure. This includes:

- Eclipse Equinox (www.eclipse.org/equinox/)

  Equinox is an implementation of the OSGI R4 core framework.

- Eclipse Communication Framework (ECF – see www.eclipse.org/ecf)

  ECF is a platform for message-oriented systems offering collaboration features.

- Corona (www.eclipse.org/corona)

  Corona is a framework for service-oriented eclipse-based environments. It supports collaboration features as well as webservice-based plugins.

3.1.2 Other supporting technologies

3.1.2.1 SOA infrastructure

Service oriented Architectures (SOA) are about to become more widespread and accepted. So let's start with a proper definition of this concept:

> *"A service-oriented architecture can be defined as a way of designing and implementing enterprise applications that deals with the intercommunication of loosely coupled, coarse grained (business level), reusable artifacts (services). Determining how to invoke these services should be through a platform independent service interface." [Wil04]*

From a common point of software development this definition is self-evident. However it isn't clearly stated how "reusable artifacts" or services should look like in detail. According to common knowledge services should be according to possibility independent from each other and provide additional specialised functionality [Bie05]. Nothing new so far. Therefore one of the important characteristics of services is their technology independence nature, the key for interoperability between different implementations on heterogeneous platforms.

Along with the architectural style SOA there are often Web Services (WS) mentioned and sometimes they are even equated. However SOA is an abstract conceptualisation and Web Services are just one possible grounding using a concrete technology, which in turn imposes technical constraints. Namely Web services are based on Internet protocols and mostly their exchanged messages are in XML. Therefore Web services use a SOA communication style while depending on concrete technology.

3.1.2.2 Service Data Objects (SDO)

SDO covers the data aspects within applications. It aims at a simplified, unified and consistent handling of such data independent of their source and format. In the end application programmers shall uniformly access and modify data from distinct and heterogeneous data sources.

SDO supports different kind of data sources like RDBMS, XML and web services. It decouples the data source and the application by offering access to the data objects disconnected from the connection of the data source. Manipulations on the disconnected data objects are however also supported. All gathered manipulations are propagated to the data source. Details of the changes are available in the form of change summaries.

It defines both a dynamic and static API to access and manipulate a graph of data objects. The static API is generated for a specific schema of data like XML schema or relational schema. Both

2006–2007 © Copyright lies with the respective authors and their institutions.

APIs allows to traverse the graph of data objects. Additionally simple queries in the form of XPath expression are possible for more complex traversal of objects.

The dynamic API has a rather complete support for accessing the meta model, which goes beyond Java reflection.

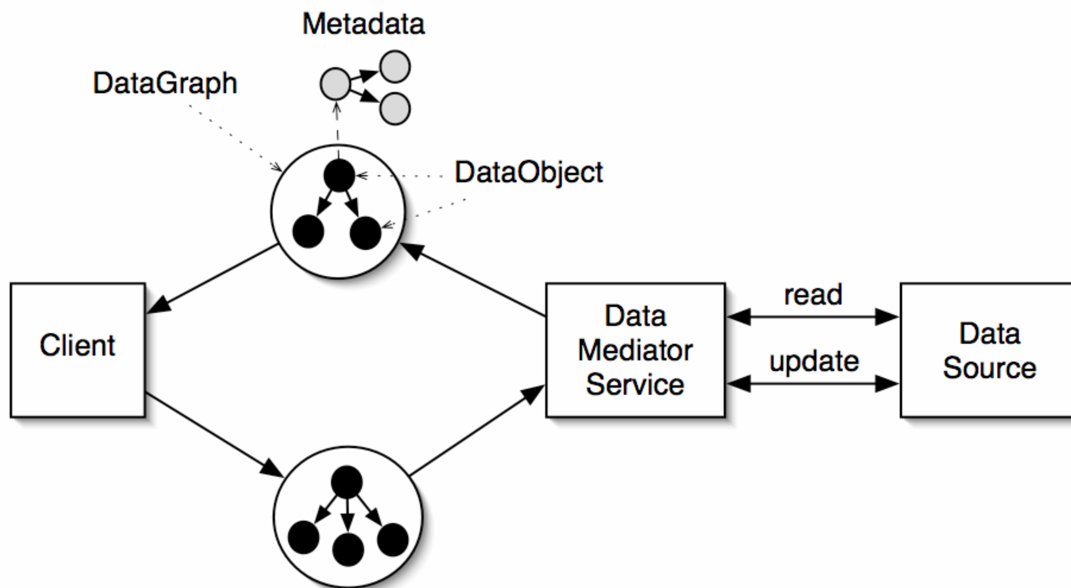

Figure 6: SDO components

Due to its disconnected mode and complete support of XML it is especially suited to access web services in a very comfortable way in Java and other programming languages. It is especially possible to handle wild card XML content with elements and attributes not specified in the XML schema.

The realisation is based on a Data Mediator Service, which is responsible for synchronizing the disconnected DataGraph of DataObjects used by the client with the data in the external data source.

3.1.3 Technologies for language processing

This section looks at architectures relevant to the networked collaborative annotation task, specifically:

- GATE, a General Architecture for Text Engineering
- UIMA, an Unstructured Information Management Architecture
- ISO TC37/SC4 and the LIRICS standardisation project

Of these GATE is most relevant to NeOn and in section 5 we will define a GATE-based distributed collaborative annotation service for the project. GATE is tracking the ISO standards and is a reference implementation for several of their components, and GATE is interoperable with UIMA. Therefore we choose GATE as the foundation of NeOn networked semantic metadata services.

3.1.3.1 GATE

GATE is one of the oldest and most widely used architectures for language engineering. This section sketches GATE and recent developments in a semantic technology context.

GATE, a General Architecture for Text Engineering, was first released in 1996, then completely re-designed, re-written, and re-released in 2002. The system is now one of the most widely-used systems of its type and is a relatively comprehensive infrastructure for language processing software development. The new UIMA architecture from IBM/Apache (see next section) has taken inspiration from GATE and IBM have paid USFD to develop an interoperability layer between the two systems. Key features of GATE are:

- Component-based development reduces the systems integration overhead in collaborative research.
- Open source, documented and supported software enhances the repeatability of experiments.
- Automatic performance measurement of Language Engineering (LE) components promotes quantitative comparative evaluation.
- Distinction between low-level tasks such as data storage, data visualisation, discovery and loading of components and the high-level language processing tasks.
- Clean separation between data structures and algorithms that process human language.
- Consistent use of standard mechanisms for components to communicate data about language, and use of open standards such as Unicode and XML.
- Insulation from idiosyncratic data formats (GATE performs automatic format conversion and enables uniform access to linguistic data).
- Provision of a baseline set of LE components that can be extended and/or replaced by users as required.

GATE is an architecture, a framework and a development environment for LE (Language Engineering) applications. As an architecture, it defines the organisation of an LE system and the assignment of responsibilities to different components. As a framework, it provides reusable implementations for LE components and a set of prefabricated software building blocks that language engineers can use, extend and customise for their specific needs. As a development environment, it helps its users minimise the time they spend building new LE systems or modifying existing ones, by aiding overall development and providing a debugging mechanism for new modules. Because GATE has a component-based model, this allows for easy coupling and decoupling of the processors, thereby facilitating comparison of alternative configurations of the system or different implementations of the same module (e.g., different parsers). The availability of tools for easy visualisation of data at each point during the development process aids immediate interpretation of the results.

Applications developed within GATE can be deployed outside its Graphical User Interface (GUI), using programmatic access via the GATE Application Programming Interface (API). In addition, the reusable modules, the document and annotation model, and the visualisation components can all be used independently of the development environment.

2006–2007 © Copyright lies with the respective authors and their institutions.

GATE is engineered to a high standard and supports efficient and robust text processing. It is tested extensively, including regression testing, and frequent performance optimization. GATE has proved capable of processing gigabytes of text and thousands of documents. The system is supported by extensive documentation, online tutorials and an established mailing list.

In the rest of this section, we provide an overview of the GATE architecture, discuss the annotation graph based data representation, and introduce important core elements, such as multilingual support, distributed resources, execution strategies, and the JAPE finite-state processing engine.

An Overview of the architecture

The GATE architecture distinguishes between data, algorithms, and their visualisation. GATE components are one of three types (each implemented as a special type of Java Bean):

- Language Resources (LRs) represent entities such as lexicons (e.g. WordNet), corpora or ontologies.
- Processing Resources (PRs) represent entities that are primarily algorithmic, such as parsers, generators or n-gram modellers.
- Visual Resources (VRs) represent visualisation and editing components that participate in GUIs.

These resources can be local to the user's machine or remote (available over the Internet), and all can be extended by users without modification to GATE itself. One of the main advantages of separating the algorithms from the data they require is that the two can be developed independently by language engineers with different types of expertise, e.g. programming and linguistics. Similarly, separating data from its visualisation allows users to develop alternative visual resources, while still using a language resource provided by GATE.

Collectively, all resources are known as CREOLE (a Collection of REusable Objects for Language Engineering), and are described in XML configuration files, which declare their name, implementing class, parameters, icons, etc. This component metadata is used by the framework to discover and load available resources. GATE offers comprehensive multilingual support using Unicode as its default text encoding. It also provides a means of entering text in various languages, using virtual keyboards where the language is not supported by the underlying operating platform (McEnery2000; Tablan2002).

When an application is developed within GATE's graphical environment, the user chooses which processing resources go into it (e.g. tokeniser, POS tagger), in what order they will be executed, and on which data (e.g. document or corpus). The application results can be viewed in the document viewer/editor. Applications can be saved, reloaded, and embedded in other systems.

Data Representation and Handling

GATE supports a variety of formats including plain text, HTML, XML, RTF, and SGML in order to enable the processing of a wide range of documents and improve interoperability with other LE infrastructures. In all cases, when a document is opened in GATE, the format is analysed and converted into a single unified model of annotation. The annotation format is a modified form of the TIPSTER format [Grishman97], which has been made largely compatible with the Atlas format [Bird200a], and is isomorphic with stand-off markup as widely adopted by the SGML/XML community (Ide2000). Interoperability with UIMA is managed via translation between the annotation graphs in GATE and the Common Analysis Structure in UIMA; these two structures are very similar (the main difference being the mandatory type system present in UIMA): see next section.

The annotations associated with each document are a structure central to GATE, because they encode the data read and produced by all processing modules, as well as the formatting information originally present in the documents.

Annotations are organised in annotation graphs, where the vertices are anchored in the document content. Annotations are the arcs in the graph; they have a start node and an end node, an identifier, a type and a set of features. Nodes have pointers into the content, e.g. character offsets for text, milliseconds for audio-visual content. There can be more than one annotation graph associated with each document. The support for separate annotation graphs (called families in IBM's TEXTRACT architecture, the predecessor of UIMA (Neff2004)) is needed to allow separation of information into different categories.

GATE has a single model for encoding information that describes documents, collections of documents (corpora), and annotations on documents, based on attribute name/attribute value pairs, called features. At present GATE uses features only as a representation formalism and does not provide operations like unification.

In order to facilitate the interpretation and validation of features associated with each annotation type, GATE provides annotation schemas, which are encoded in the XML Schema language supported by W3C, and define all attributes and the type of their value (e.g., orgType for Organization annotations). If the value needs to be one of a predefined set of values (e.g., a part-of-speech tagset), then this set of values is also given in the annotation schema. GATE's annotation schemas were inspired by TIPSTER's annotation type declarations (Grishman 97), however GATE also uses them to facilitate and validate users' input during manual annotation. GATE schemas are optional, unlike UIMA's type system which is mandatory (see below).

JAPE: Finite State Processing Over Annotations

GATE has facilities for finite state processing over annotations based on regular expressions. JAPE, a Java Annotation Patterns Engine, is similar to the finite state pattern matching transducer part of the TEXTRACT architecture (Neff et al.; unfortunately this software is not part of UIMA). A JAPE grammar consists of a set of phases, each of which consists of a set of pattern/action rules. The phases run sequentially and constitute a cascade of finite state transducers over annotations. This cascade is similar to the finite state cascade used for parsing by (Abney1996) in that rules in the current phase can refer to annotations created in earlier phases and there is no recursion.

The left-hand-side (LHS) of the rules consist of an annotation pattern (or a macro describing such a pattern), that may contain Kleene regular expression operators (e.g. *, ?, +). The right-hand-side (RHS) consists of annotation manipulation statements (in JAPE or Java). Annotations matched on the LHS of a rule may be referred to on the RHS by means of labels that are attached to pattern elements. At the beginning of each grammar, several options can be set:

- Control: this defines the method of rule matching: brill, appelt, or first. The brill style means that when more than one rule matches the same region of the document, they are all fired. With the appelt style, only one rule can be fired for the same region of text (the choice depends on the length of the matching region and rule priority). With the first style, a rule fires for the first match that is found and no other matching attempts are made for that region of the document.

- Input annotation types: restrict the input to the grammars in order to improve execution speed and control the range for matching in the document. The example below shows a JAPE rule with pattern and corresponding action. (On the LHS, each such pattern is enclosed in a set of round brackets and has a unique label; on the RHS, each label is associated with an action.) In this example, the Lookup annotation is labelled jobtitle and

2006–2007 © Copyright lies with the respective authors and their institutions.

is given the new annotation JobTitle; the TempPerson annotation is labelled person and is given the new annotation Person.

```
Rule: PersonJobTitle
  Priority: 20
(

):jobtitle
(

):person
-->
  :jobtitle.JobTitle =,
  :person.Person =
```

A large proportion of the processing resources bundled with GATE use JAPE.

Execution strategies

Execution strategies in GATE determine how the algorithms (i.e. processing resources) are combined to form a complete application and how that application is executed on the provided input (i.e. language resources such as corpora). These execution strategies are implemented as *controllers* and the user chooses the one most suitable for their application.

The two main execution strategies commonly found in LE applications are pipeline (i.e. sequential) and parallel. For example, GATE's pipeline controller executes the PRs in the specified order and with the given parameters (e.g. a document). A variation of this controller is the corpus pipeline, which executes the PRs sequentially over a given corpus, taking care of loading the documents from the disk (if necessary), processing each of them, and saving the results.

At present GATE does not provide a controller which supports parallel execution of PRs, although it is planned for future work. However, it does support execution of applications on different machines over data shared on the server, which enables simultaneous processing of documents.

Other types of execution strategies are conditional (equivalent to an if statement in programming languages) and iterative (equivalent to loop statements). The conditional controller was introduced in order to allow flexible processing of heterogeneous data (e.g., a corpus consisting of documents from several domains, styles, and genres). The controller allows the user to specify for each PR the conditions when it can be executed: always; never; or conditionally - only if a particular feature with a given value is present on the document.

For example, a categorisation module that always fires can assign different genres to each document as features (e.g., sport, political, business). Then the application can contain PRs trained specifically for the given type of documents and at execution time, based on the document type, the controller will determine which PRs are executed. For example, a text about sport would use a different named entity recogniser from the one about politics (England is a team in the first case and a location in the second) or a text in all uppercase would use a different POS tagger (amongst other things) from a text in mixed case. However, the application will list all alternative PRs (e.g. two POS taggers and 3 entity recognisers) but only those appropriate for the current document will be executed on it, whereas a different set of PRs might be executed on the next one.

3.1.3.2 UIMA

UIMA, the Unstructured Information Management Architecture, provides a subset of GATE's features and also implements Enterprise Applications Integration routes for middleware technology

such as WebSphere. This section introduces UIMA and describes the GATE/UIMA interoperation layer. (It is adapted from documents of the OASIS/Open UIMA standardisation committee, of which the University of Sheffield is a founder member.)

UIMA is a platform for language processing developed by IBM. It has many similarities to the GATE architecture – it represents documents as text plus annotations, and allows users to define pipelines of analysis engines that manipulate the document (or Common Analysis Structure in UIMA terminology) in much the same way as processing resources do in GATE. IBM has donated an implementation of the UIMA architecture called the UIMA SDK to the Apache Software Foundation. This provides support for building analysis components in Java and C++ and running them either locally on one machine, or deploying them as services that can be accessed remotely. The SDK is available for download from Apache.

UIMA builds on the work of prior IBM researchers and projects dedicated to advancing the state of the art in frameworks for text and multimodal analytics including TAF, TALENT (TAL1) and WebFountain (WF1, WF2). It has been inspired and influenced by other projects outside of IBM including TIPSTER (TIP1), Mallet (MAL1), GATE (GATE1, GATE2), OpenNLP (ONLP1), Atlas (Laprun1) and Catalyst (CAT1).

UIMA formally began at IBM in 2001 as a project dedicated to enabling interoperability of analytics across research projects underway in different geographies across the globe by scores of researchers and to facilitate their integration and deployment into a variety of IBM Information Management products. The technical goals were to provide a common software framework that would facilitate the creation, discovery, integration and deployment of component text and multimodal analytics.

The immediate problem motivating the project was that numerous efforts throughout IBM Research and IBM Software Group were focused on creating independent unstructured information analytics based on different technologies, interfaces, implementations and data representations. These independently developed analytics included for example: tokenization, language identification, document structure parsing, grammatical parsing, named-entity detection, chemical name and relationship detection, summarization, document classification, topic detection and tracking, speaker identification, speech transcription, natural language translation, image analysis, video object detection and tracking etc. Complete solutions or end-user applications often required a combination of numerous highly specialized capabilities. The technical hurdles involved in reusing, combining and deploying these independently developed analytics across research projects and product platforms were often prohibitive. The results were inefficiencies and/or sub-optimal analytic performance.

In late 2004 IBM released the UIMA Software Developers Kit (SDK) on IBM alphaWorks (http://www.ibm.com/alphaworks/tech/uima). The SDK is freely available and provides the tools and run-time necessary for creating, composing and deploying component analytics. These analytics may be implemented by the developer to analyze and assign semantics to multi-modal data including, for example, combinations of text, audio or video. The SDK includes a semantic search engine based on the XMLFragment query language (XMLFrags1) for users to experiment with a search facility designed to exploit the semantic elements produced by a workflow of analytics.

Since 2004 industrial, academic and government research & development projects inside and outside of IBM have applied the UIMA SDK as a foundation for building and/or enhancing applications that process unstructured information. Feedback from users has helped inform the ideas presented in this report.

In early 2006 IBM contributed an implementation of the UIMA Framework to the open-source community through source forge (UIMASrc1). The source includes everything in the SDK except the XMLFragment search engine. In 4Q 2006, the open-source framework was accepted as an Apache incubation project, where IBM and non-IBM committers will participate in its collaborative

2006–2007 © Copyright lies with the respective authors and their institutions.

development. (ApacheUIMA1). Throughout this report we use the term Apache UIMA to refer to this implementation.

The Apache UIMA implementation requires application and programming commitments at the Java API level to provide a rich set of functions and to facilitate high-degrees of component-level interoperability. Any given framework implementation, however, may not satisfy the requirements of all UIMA applications. Some, for example, may require very lightweight browser-based analytics or a very different programming model, while others may require heavyweight, carefully managed, highly scalable solutions and yet others may depend on legacy infrastructure or middleware.

This diverse variety of potential implementations suggests that there should be a more general specification for interoperability that may allow for different framework implementations and different levels of compliance facilitating interoperability for a broader range of application and programming requirements.

To help define a broader, platform independent standard that can guide the open-source collaborative development of Apache UIMA and other related frameworks, applications and tools while maintaining broad interoperability, IBM has convened a Technical Committee to develop a standard specification under the auspices of OASIS (www.oasis-open.org) a Standards Development Organization. The intent is that such a standard would allow different frameworks to emerge, while also allowing applications built on different platforms and programming models to have a standard means to share analysis data and analytic services. Such a standard would lower the barrier for getting analytics to interoperate, allowing a broader community to discover, reuse and compose independently-developed text and multi-modal analytics in UIMA applications. USFD is a founder and proposing member of this committee, and will make its results available within NeOn.

There are eight elements of UIMA that will be covered by the standardisation activity. They are listed in brief below.

1. Common Analysis Structure (CAS) Specification. Provides a simple and extensible model for representing analysis data as a standard object model that may be easily instantiated and manipulated in object oriented programming systems. This element of the specification is provided as a UML (UML1) model. We propose adopting the XML Metadata Interchange (XMI) specification (XMI1, XMI2) to provide a standard means for representing analysis data as an XML document.

2. Type-System Language Specification. Provides a standard means for associating object model semantics with artifact metadata that complies with object modeling standards. The proposal is to use Ecore as the type system language. Ecore is the modeling language used in the Eclipse Modeling Framework (EMF1) and is tightly aligned with the OMG's EMOF standard1.

3. Type-System Base Model. Provides a standard and extensible set of domain-independent types generally useful for analyzing unstructured information. For example we define a type Annotation to represent objects that have regional references (e.g., offsets) into the value of an attribute of another object. It is intended that annotations describe or "annotate" the unstructured content in these values.

4. The behavioral Metadata Specification. Provides a standard declarative means for describing the capabilities of analysis operations in terms of what types of CASs they can process, what elements in a CAS they can analyze, and what sorts of effects they would have on CAS contents as a result. While we do not provide a complete and formal specification for behavioral Metadata, we provide a discussion of requirements and a rough proposal appealing to the OCL standard (OCL1).

5. Processing Element Metadata Specification. Provides a standard declarative means for describing identification, configuration and behavioral information about Processing Elements (analytics and flow controllers). This specification is represented as a UML Model, and we use the XMI standard to represent the processing element metadata as XML. This section of the

specification refers to the behavioral Metadata Specification to represent a processing element's behavioral information.

6. Abstract Interfaces. Abstractly describes the interfaces to the two different types of Processing Elements, namely, Analytics and Flow Controllers. These abstract interfaces are specified with a UML model.

7. WSDL Service Descriptions. Provides a standard means for describing Processing Elements as web services using WSDL (WSDL1). We also define a standard SOAP binding.

8. Aggregate Analytic Descriptor Specification. Provides a standard declarative means for an aggregate analytic to

- refer to its constituent analytics
- identify a flow controller which determines the order in which the constituent analytics of the aggregate are invoked on a CAS.
- define mappings to facilitate the composition of independently-developed analytics.

3.1.3.3 ISO TC37/4

We can distinguish two types of standardisation: de facto and de jure. This section covers the current de jure process active at ISO (in Technical Committee 37, Sub-Committee 4), but it should first be noted that the field is more strongly influenced by de facto standardisation and that the practical implementations represented by GATE and UIMA (see above) are more probably more promising starting points for the NeOn collaborative annotation architecture.

The objective of ISO/TC 37/SC 4 is to prepare various standards by specifying principles and methods for creating, coding, processing and managing language resources, such as written corpora, lexical corpora, speech corpora, dictionary compiling and classification schemes. These standards will also cover the information produced by natural language processing components in these various domains. Standards produced by ISO/TC 37/SC 4 should particularly address the needs of industry and international trade as well as the global economy regarding multi-lingual information retrieval, across cultural technical communication and information management.

The goal of ISO/TC 37/SC 4 is also to ensure that new developments in language engineering, knowledge management and information engineering satisfy the norms of international standardization for:

- development standards and related documents to maximize the applicability of language resources,
- relating the language resources of different kinds to their applications, and
- enhancing the application of recognized methods and tools in language resources.

USFD is a participant in ISO TC37/SC$ both directly and via the LIRICS project. LIRICS addresses the needs of today's information and communication society where globalisation and localization necessitate multilingual communication creating an increasing need for new standardization as well as urgent recognition of existing de facto standards and their transformation into 'de jure' International Standards. LIRICS thus aims to:

- Provide ISO ratified standards for language technology to enable the exchange and reuse of multilingual language resources;

2006–2007 © Copyright lies with the respective authors and their institutions.

- Facilitate the implementation of these standards for end-users by providing an open-source implementation platform, related web services and test suites building on legacy formats, tools and data;
- Gain full industry support and input to the standards development via the Industry Advisory group and demonstration workshops
- Provide a pay-per-use business model for use by Industry and in particular SMEs validated during the project for the benefit of all actors in the content and language industries

The LIRICS open source reference implementation for the new ISO standards uses GATE for annotation services.

The LIRICS Consortium brings together leading experts in the field of NLP and related standards development via participation in ISO committee and National Standardisation committees. The Consortium has strong Industry support and involvement through the 21 members of the LIRICS Industry Advisory Group.

The standards development work in LIRICS starts from an analysis of existing and emerging formats for the standards in the project scope. From this analysis the partners have defined metamodels and data categories for lexica, morpho-syntactic annotation, syntactic annotation and semantic content following closely the procedures established by ISO, as well as an on-line environment for accessing and maintaining data categories.

The deliverables are direct inputs to the ISO committee ballots. A quality assurance procedure within the project by partners on the ISO committee ensures acceptance of the proposed standards at ISO level.

The implementation platform and the web services offered will be the basis for the LIRICS exploitation plan enabling a new pay-per-use business model for language technology actors. All these activities will be supported via a strong dissemination activity targeting end-users of LIRICS standards at demonstration workshops and a specific training session on NLP standards for partners of eContent projects.

LIRICS is delivering a portfolio of standards for ratification by ISO including:

- a metamodel for lexical representation;
- metamodels and data categories for morpho-syntactic and syntactic annotation;
- reference data categories for semantic annotation;
- test suites in nine European languages;
- an open-source implementation platform, compatible with major legacy systems and tools.

LIRICS will increase awareness of language engineering standards and promote their take-up on a European scale.

3.2 Existing Infrastructures

3.2.1 Components of existing Ontology Engineering Tools

In this section we provide an overview of state-of-the-art reasoners and repositories. We restrict the overview to systems that provide support for ontology languages relevant to the NeOn networked ontology model, i.e. in particular OWL, but also rule languages.

The line between reasoners and repositories for ontologies is not a strict one to draw. Often, ontology management systems provide support for both in some integrated form. Still, a distinction is useful, as reasoners and repositories aim at different goals:

While repositories aim at being able to efficiently store and retrieve large collections of data (i.e. managing explicit facts), reasoner focus on deduction procedures to derive implicit knowledge.

Ontology repositories often provide database-like functionalities, sometimes provide (typically limited) inferencing /reasoning support, but often allow to integrate with other, external reasoners. In turn, many reasoners either also exhibit repository functionality or provide means to access repositories (e.g. in a file system or relational database).

In the following, we start with an overview of existing reasoners, where we discuss the supported ontology languages, their reasoning approach, availability and interfaces. The overview is partially based on the Description Logic Reasoner site of Ulrike Sattler. (http://www.cs.man.ac.uk/~sattler/reasoners.html)

- Cerebra Engine is a commercial C++-based reasoner. It implements a tableau-based decision procedure for general TBoxes (subsumption, satisfiability, classification) and ABoxes (retrieval, tree-conjunctive query answering using a XQuery-like syntax). It supports the OWL-API and comes with numerous other features.

- FaCT++ is a free open-source C++-based reasoner for SHOIQ with simple datatypes (i.e., for OWL-DL with qualifying cardinality restrictions). It implements a tableau-based decision procedure for general TBoxes (subsumption, satisfiability, classification) and incomplete support of ABoxes (retrieval). It supports the lisp-API and the DIG-API.

- KAON2 is a free (free for non-commercial usage) Java reasoner for SHIQ extended with the DL-safe fragment of SWRL. It implements a resolution-based decision procedure for general TBoxes (subsumption, satisfiability, classification) and ABoxes (retrieval, conjunctive query answering). It comes with its own, Java-based interface, and supports the DIG-API.

- OntoBroker is a commercial Java based main-memory deductive database engine and query interface. It processes F-Logic ontologies and provides a number of additional features such integration of relational databases and various built-ins. The new version of OntoBroker operates on the KAON2 API.

- Pellet is a free open-source Java-based reasoner for SROIQ with simple datatypes (i.e., for OWL 1.1). It implements a tableau-based decision procedure for general TBoxes (subsumption, satisfiability, classification) and ABoxes (retrieval, conjunctive query answering). It supports the OWL-API, the DIG-API, and Jena interface and comes with numerous other features.

2006–2007 © Copyright lies with the respective authors and their institutions.

- RacerPro is a commercial (free trials and research licenses are available) lisp-based reasoner for SHIQ with simple datatypes (i.e., for OWL-DL with qualified number restrictions, but without nominals). It implements a tableau-based decision procedure for general TBoxes (subsumption, satisfiability, classification) and ABoxes (retrieval, nRQL query answering). It supports the OWL-API and the DIG-API and comes with numerous other features.

- OWLIM is semantic repository and reasoner, packaged as a Storage and Inference Layer (SAIL) for the Sesame RDF database. OWLIM uses the TRREE engine to perform RDFS, and OWL DLP reasoning. It performs forward-chaining of entailment rules on top of RDF graphs and employs a reasoning strategy, which can be described as total materialization. OWLIM offers configurable reasoning support and performance. In the "standard" version of OWLIM (referred to as SwiftOWLIM) reasoning and query evaluation are performed in-memory, while a reliable persistence strategy assures data preservation, consistency and integrity.

	Cerebra	FACT++	KAON2	Pellet	RacerPro	OntoBroker	OWLIM
Interfaces	OWL API	DIG	KAON2 API (OWL part)	OWL API, DIG, Jena API	OWL API, DIG	KAON2 API (F-Logic part)	Sesame API
Reasoning Approch	tableaux	Tableaux	resolution	tableaux	tableaux	Prolog	Forward chaining rules
Supported Logic	Close to OWL DL	SHOIQ	SHIQ + DL-safe rules	SROIQ + DL-safe rules	SHIQ	F-Logic	OWL DLP
Based on	C++	C++	Java	Java	Lisp	Java	Java

In the following we additionally discuss two implementations of ontology repositories: Jena and Sesame are the two most popular implementations of RDF stores. They play a separate role, as their primary data model is that of RDF. However, they deserve discussion, as they offer some OWL functionalities and limited reasoning support.

- Sesame (http://openrdf.org) is an open source repository for storing and querying RDF and RDFS information. OWL ontologies are simply treated on the level of RDF graphs. Sesame enables the connection to DBMS (currently MySQL, PostgreSQL and Oracle) through the SAIL(the Storage And Inference Layer) module, and also offers a very efficient direct to disk Sail called Native Sail. Sesame provides RDFS inferencing and allows querying through SeRQL, RQL, RDQL and SPARQL. Via the SAIL it is also possible to extend the inferencing capabilities of the system. (In fact, this is how the

OWLIM reasoner is realized.) The main ways to communicate with the Sesame modules are through the Sesame API or through the Sesame Server, running within a Java Servlet Container.

- Jena is a Java framework for building Semantic Web applications (http://jena.sf.net). It offers the Jena/db module which is the implementation of the Jena model interface along with the use of a database for storing/retrieving RDF data. Jena uses existing relational databases for persistent storage of RDF data; Jena supports MySQL, Oracle and PostgreSQL. The query languages offered are RDQL and SPARQL. Just as in Sesame, the OWL support is realized by treating OWL ontologies as RDF graphs. However, in addition Jena also provides a separate OWL API and allows to integrate with external reasoners, such as Pellet.

3.2.1.1 OntoStudio

The starting point for the NeOn toolkit is the ontology engineering platform of ontoprise, OntoStudio. OntoStudio is a the front-end counterpart to OntoBroker, a very fast datalog based FLogic inference machine. Consequently a focus of the OntoStudio development has been on the support of various tasks around the application of rules. This includes the direct creation of rules (via a graphical rule editor) but also the application of rules for the dynamic integration of datasources (using a database schema import and a mapping tool). The coming version of OntoStudio will have additional support for rule creation and management such as a rule debugger and a textual rule editor for F-Logic including features like auto-completion and syntax-check (based on an incremental parsing). All features will be available for the NeOn toolkit (either as open-source or closed-source commercial extensions).

OntoStudio is available with a main memory- or database-based model, is therefore scaleable and is thus suitable for modelling even large ontologies. Based on Eclipse OntoStudio provides an open framework for plugin developers. It already provides a number of plugins such as a query plugin, a visualizer and a reporting plugin supporting the Business Intigelligence Reporting Tool (BIRT). Along with the Eclipse philosophy “everything is a plugin” OntoStudio is highly modular. A central datamodel plugin is the entry point (and a minimal requirement) for every plugin.

OntoStudio and OntoBroker are produced and distributed by Ontoprise GmbH (Germany). It is the successor of OntoEdit, which was distributed about 5000 installations. OntoStudio, a core product of Ontoprise, plays a central role in the product family. Professional support is available.

Some characteristics are given below:

- **knowledge model as is**: tightly coupled to F-Logic (resp. its proprietary XML serialization OXML); import and export to OWL/RFD is restricted mainly to concepts which can be expressed in F-Logic. Despite some minor syntactical details the Ontoprise FLogic dialect conforms semantically to the F-Logic definition [Kifer1995]. Ontoprise is in close contact with the F-Logic forum to work on future versions of F-Logic and further standardization efforts.

- **supported reasoners**: currently OntoBroker; in future also KAON2 (see below);

- **supported languages**: currently native Frame-Logic support and subsets of OWL and RDF(S) via import/export; in future native OWL (DL) support will be provided also as part of the NeOn toolkit developments (see below);

2006–2007 © Copyright lies with the respective authors and their institutions.

- **platforms**: Windows and Linux;
- **performance and connectivity**: designed for high scalability; connectors to Oracle, MySql, DB2; works also with large ontologies;
- **license**: free for noncommercial users; the editor platform (without reasoner and a number of commercial plugins) will be made available under GPL;
- **addresses**: domain experts with a basic understanding of ontologies and rules (OntoStudio encapsulates language syntax apart from textual editor);
- **team support** [workflow; collaboration; documentation; versioning]: currently none, but planned as part of the NeOn toolkit (see below);
- **GUI / visualization**: concept taxonomy; textual and graphical formula editor; instance view, graphical ontology visualizer, query tabs etc.

Extension Points

Some of the additional extension-points are provided to modify the main component in OntoStudio, the Ontology Navigator.

The Ontology Navigtor is a completely modifiable and extensible view on (not necessarily) ontology elements. The developer can show additional elements (almost everywhere) in the tree, can define specific drag and drop operations and gets support for the definition of additional context menu entries.

Another main component in OntoStudio is the Entity Properties View which shows property pages for the elements in the user interface. Additional property pages can be integrated in this view by the use of the corresponding extension point.

The NeOn Approach

Many of the OntoStudio capabilities are in line with the NeOn approach and fulfil the corresponding requirements. At the same time some very basic aspects of the NeOn toolkit are missing. The following list is a high-level summary of the most important points rather than a detailed mapping to requirements.

✓ OntoStudio is a modular and extensible platform.

✓ OntoStudio has extensive rule support.

✓ OntoStudio has integration capabilities for “non semantic” technology.

✗ OntoStudio does not offer native OWL/DL support.

✗ OntoStudio does not provide lifecycle support.

✗ OntoStudio does not have collaboration capabilities.

3.2.1.2 Protégé 3.2 beta

Protégé 3.2 beta is the latest version of the Protégé OWL editor, created by the Stanford Medical Informatics group at Stanford University. Protégé 3.2 beta is a Java-based open source standalone application to be installed and run in a local computer. It enables users to load and save OWL and

RDF ontologies, edit and visualize classes, properties and SWRL [SWRL2004] rules, define logical class characteristics as OWL expressions, edit OWL individuals for Semantic Web markup.

Protégé as a hybrid tool

With respect to the supported languages Protégé is a hybrid tool. The internal storage format of Protégé is frame-based. Therefore Protégé has native frame-support, based on CLIPS [REF]. The support for OWL is provided by a special plugin that fits into the Protégé plugin architecture (see below).

Plugin Architecture

OWL Support

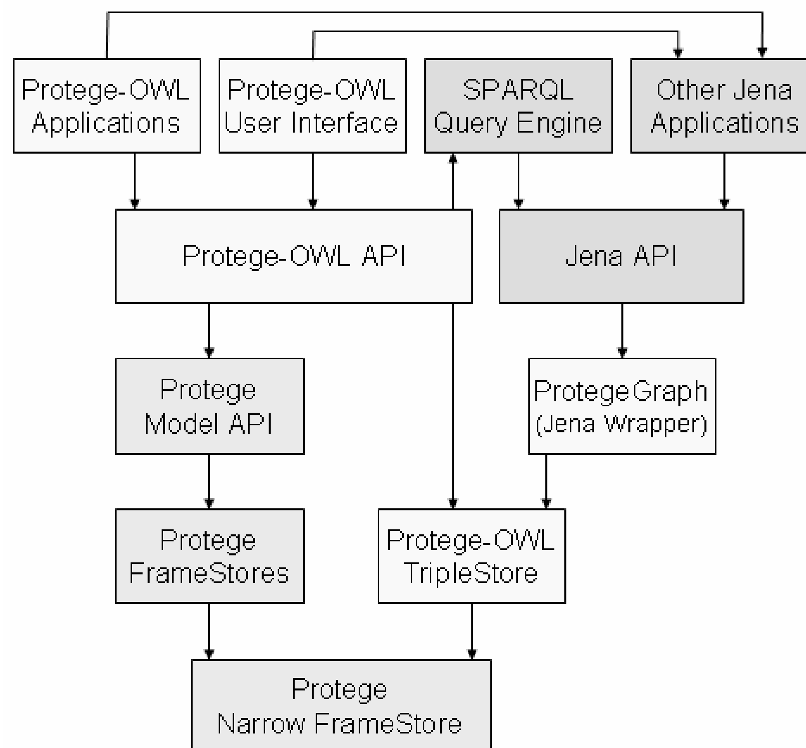

Figure 7: Architecture of the Protégé OWL

The Protégé-OWL API is an open-source Java library for OWL and RDF(S). The API provides classes and methods to load and save OWL files, to query and manipulate OWL data models, and to perform reasoning based on Description Logic engines. The API is designed to be used in two contexts: (1) development of components that are executed inside the Protégé UI, and (2) development of stand-alone applications (e.g., Swing applications, Servlets or Eclipse plugins).

As depicted in the above figure, the OWL APIs implementation rely both on the frame-based knowledge base for low level (file or DBMS based) triple storage, and both on the Jena[3] APIs for various services, such as OWL parsing and datatype handling.

[3] Jena, developed by HP Labs, is one of the most widely used Java APIs for RDF and OWL (http://jena.sourceforge.net/).

2006–2007 © Copyright lies with the respective authors and their institutions.

The Protégé-OWL API can be used to generate a Jena Model at any time in order to query the OWL model, for example by means of the SPARQL RDF query language [SPARQL]. Reasoning can be performed by means of an API which employs an external DIG[4] compliant reasoner, such as one of the following:

- RACER (http://www.sts.tu-harburg.de/%7Er.f.moeller/racer/)
- FaCT (http://www.cs.man.ac.uk/%7Ehorrocks/FaCT/)
- FaCT++ (http://owl.man.ac.uk/factplusplus/)
- Pellet (http://www.mindswap.org/2003/pellet/)

The OWL APIs allows only for accessing OWL models stored in files. There are however cases where W3C conformant OWL files produced by other tools cannot be read.

By accessing the APIs of the RDF(s)-DB Backend Plugin[5] it is be possible to store and load models from a Sesame repository. However, the development of the RDF(s)-DB Backend Plugin is currently discontinued.

The OWL APIs architecture doesn't envisage itself any "built-in" extension facility; nonetheless, the Protégé plug-in system could allow for some customization or extension of the APIs behaviour. Protégé features a client/server multiuser architecture by means of an RMI server: http://protege.cim3.net/cgi-bin/wiki.pl?MultiUserTutorial. Anyway, there isn't any Web Services (Soap, Rest or XML-RPC) interface.

The features of Protégé 3.2.

- Support for frames and OWL: Protégé was designed and developed as an editor for frame-based knowledge-bases.
- Standardized components: The main components of Protégé 3.2 beta are Metadata, OWL Classes, Properties, Individuals, Forms. The editor browses and edits the ontology's class taxonomy using a tree structure. It is very easy to check and change every tab.
- Usability features: It provides common copy&paste and drag&drop functions, and also different pop-up menus according to the type and features of the ontology component being edited. Moreover there are some wizards to help user.
- Extensibility: Using the plug-ins more functions to the environment can be added, In terms of Type the plug-ins of Applications, Backends, Import/Export, Project, Slot Widgets and Tab Widgets can be attached.

[4] The DIG Interface (http://dig.sourceforge.net/) is a standardised XML interface to Description Logics systems developed by the DL Implementation Group (http://dl.kr.org/dig/).

[5] Protégé RDF(s)-DB Backend Plugin: http://protege.stanford.edu/plugins/rdfs-db/.

- Readable by DL reasoner supporting DIG: such as RACER, FaCT. It is simple to check consistency, classify taxonomy and compute inferred types.

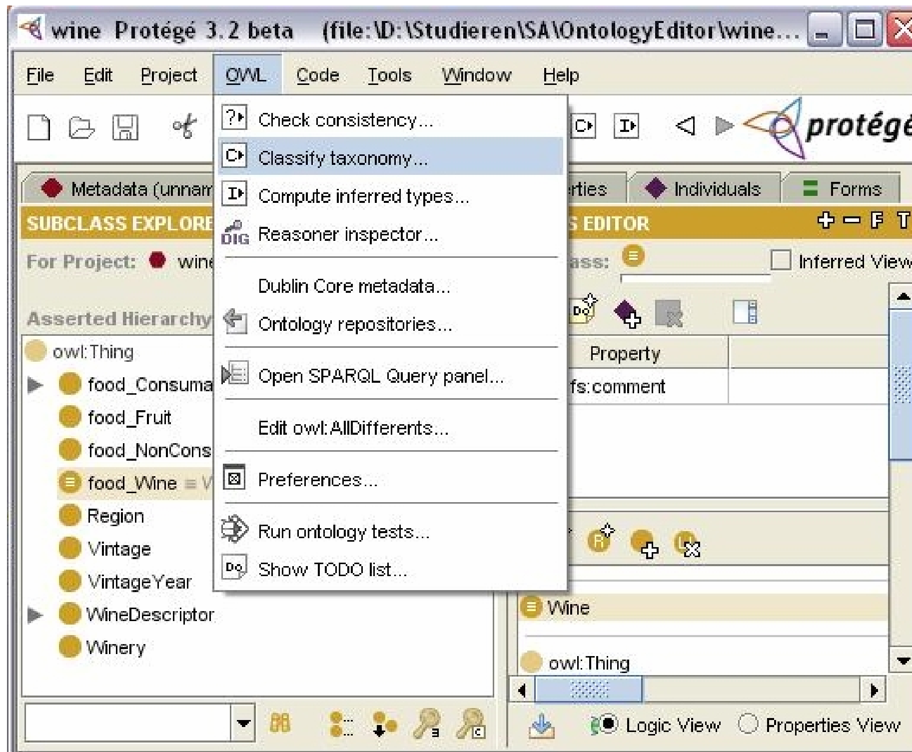

Figure 8: Protégé 3.2 Beta

3.2.1.3 Altova SemanticWorks 2006

Altova SemanticWork 2006 is the latest version of the OWL ontologies editor, created by Altova company. It is commercial software. It is visual based RDF/OWL editor that helps user visually design Semantic Web instance documents, vocabularies, and ontologies, then export them in either RDF/XML or N-Triples format. The following are its advantages.

- Visualization: In SemanticWorks, documents can be created and edited graphically in SemanticWorks's RDF/OWL View
- Multiform Interface: SemanticWorks offers ontology editing capability in a graphical user interface and in a text interface.
- Including Syntax checking and Semantics checking: Syntax checking can be carried out for RDF Schema, OWL Lite, OWL DL, and OWL Full ontologies. Semantics checking can be carried out on OWL Lite and OWL DL documents. These checking can be made while editing, thus enabling user to easily maintain the validity of the ontology.

2006–2007 © Copyright lies with the respective authors and their institutions.

- Separate tabs: The components of SemanticWorks are Classes, Properties, Instances, allDifferent and Ontologies. They can be viewed in separate tabs. The screenshot on the following figure shows that.

- Import: Imports can be reloaded at the click of a button whenever required.

- Export: Ontologies can be saved as .rdf, .rdfs, or .owl files and can be exported in their RDF/XML and N-Triples formats.

- Multiple windows: Multiple ontologies can be edited concurrently in multiple windows.

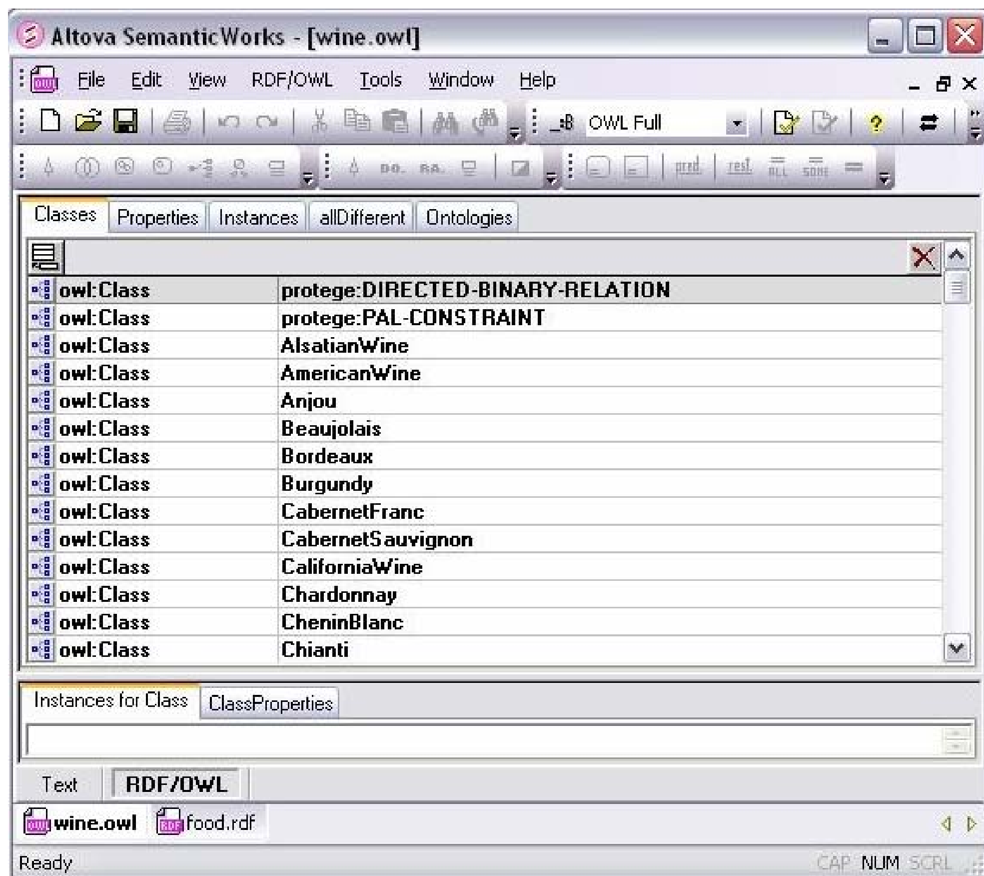

Figure 9: Altova SemanticWorks™

3.2.1.4 TopBraid Composer

TopBraid Composer is a modelling tool for the creation and maintenance of ontologies. It is a complete editor for RDF(S) and OWL models, as well as a platform for other RDF-based components and services. TopBraid Composer is built upon the Eclipse platform and uses Jena as its underlying API. Following is the advantage of it.

- Consistency checking and debugging: TopBraid Composer uses OWL Description Logic to run logical consistency checks and to classify classes and instances. The system has the open-source DL reasoner Pellet built-in as its default inference engine, but other classifiers can be accessed via the DIG interface.

- Rule: Beside its Description Logics support, TopBraid Composer can also handle traditional rule bases. The system supports rules in either the Jena Rules format or SWRL. Both types of rules are executed with the internal Jena Rules engine to infer additional relationships among resources. Rules can be edited with support of auto-completion and syntax checking.
- Visual Graphs: TopBraid Composer can visualize arbitrary relationships among RDFS/OWL resources in a graphical format.
- Eclipse plugin: This means that Composer is built on a solid platform with a large user base. Many other Eclipse plugins for editing other languages such as UML and XML exist, and therefore users can use a single tooling environment for many different modelling tasks.
- Simple to import and export: Just to choose the local file (for example file with ending .owl) it will be opened in multiple windows. That is , user can compose several files at the same time.
- Navigator: With it user can choose the file more easily.
- Namespace management: TopBraid Composer has specific mechanisms to manage namespaces and imports in the ontology documents.

You can find the features clearly by screenshot of the figure below.

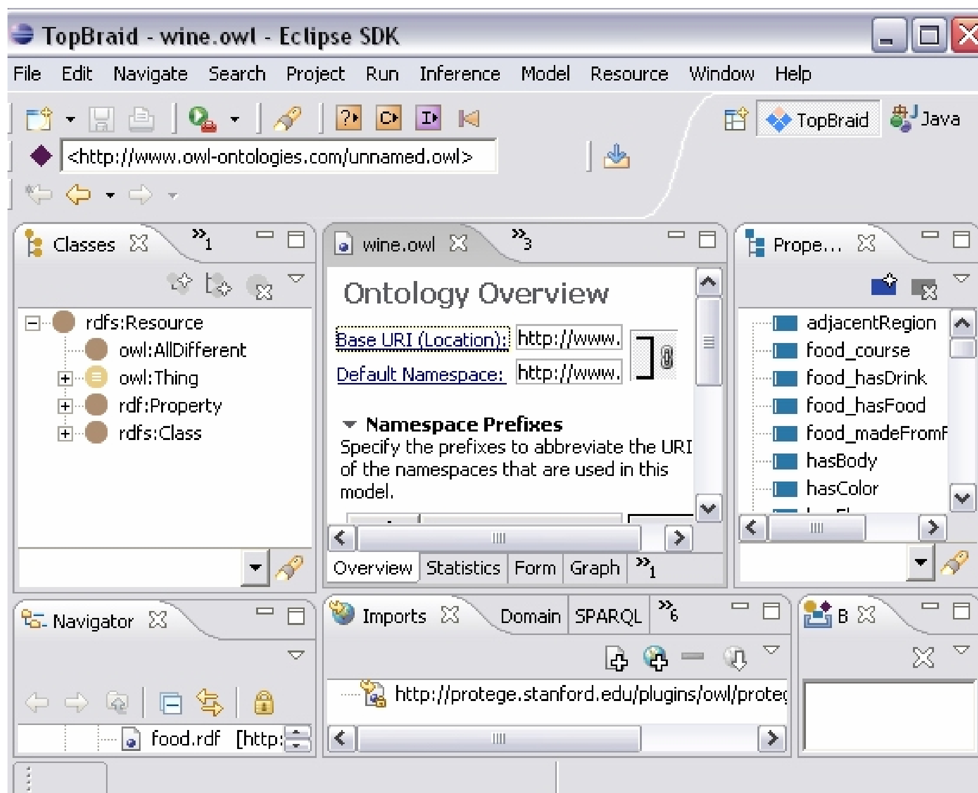

Figure 10: TopBraid Composer

2006–2007 © Copyright lies with the respective authors and their institutions.

3.2.1.5 SemTalk 2

SemTalk 2 is an editor for Semantic Web ontologies and processes. Because Microsoft Visio is embedded in SemTalk 2 it combines the graphical strength of Microsoft Visio and consistency of a professional modelling tool. To work with OWL user should use the OWL Template. Following is the advantage of it.

- Graphical: SemTalk allows user to model concept models in a structured way, based on the graphical user interface of Microsoft Visio. Visio shapes help for a better understanding of these models.

- Mircrosoft Visio is embedded: Supporting for and combination with ontology is the development trend of many software. Such as: Acrobat comes up with support for RDF Meta data. MS Word has an integrated XML editor which allows structuring documents and saving the information separately.

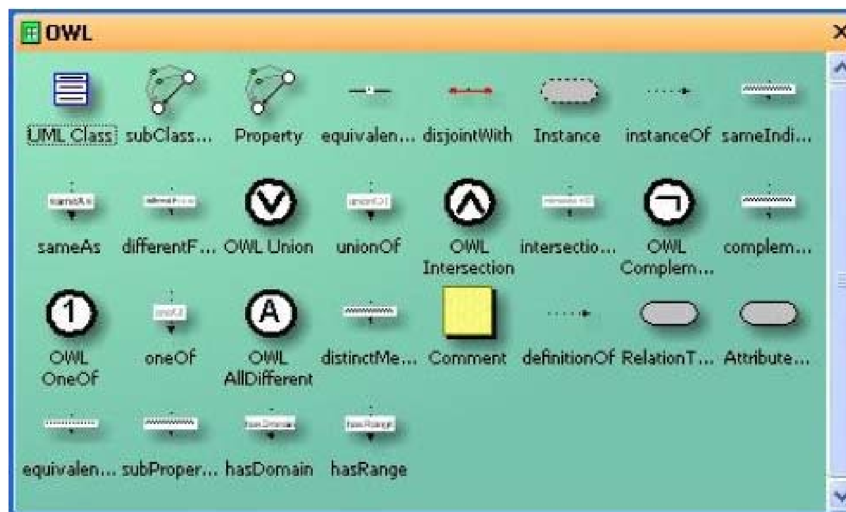

Figure 11: SemTalk

3.2.1.6 Integrated Ontology Development Toolkit (IODT)

IODT is a toolkit for ontology-driven development, created by IBM. This toolkit includes EMF Ontology Definition Metamodel (EODM), EODM workbench, and an OWL Ontology Repository (named Minerva). EODM is derived from the OMG's Ontology Definition Metamodel (ODM) and implemented in Eclipse Modeling Framework (EMF).In oder to facilitate software development and execution, EODM includes RDFS/OWL parsing and serialization, reasoning, and transformation between RDFS/OWL and other data-modeling languages. These functions can be invoked from the EODM Workbench or Minerva. EODM Workbench is an Eclipse-based, integrated, ontology-engineering environment that supports ontology building, management, and visualization.

Minerva is an OWL ontology storage, inference, and query system based on RDBMS (Relational Database Management Systems). It supports DLP (Description Logic Program), a subset of OWL DL. The system architecture of IODT is shown in the figure below.

EODM Workbench is an Eclipse-based editor for users to create, view and generate OWL ontologies. It has UML-like graphic notions to represent OWL class,restriction and property etc. EODM Workbench built by using EODM, EMF, Graphic Editing Framework (GEF), which provides the foundation for the graphic view of OWL. It also provides two hierarchical views for both OWL class/restriction and OWL object/datatype property.

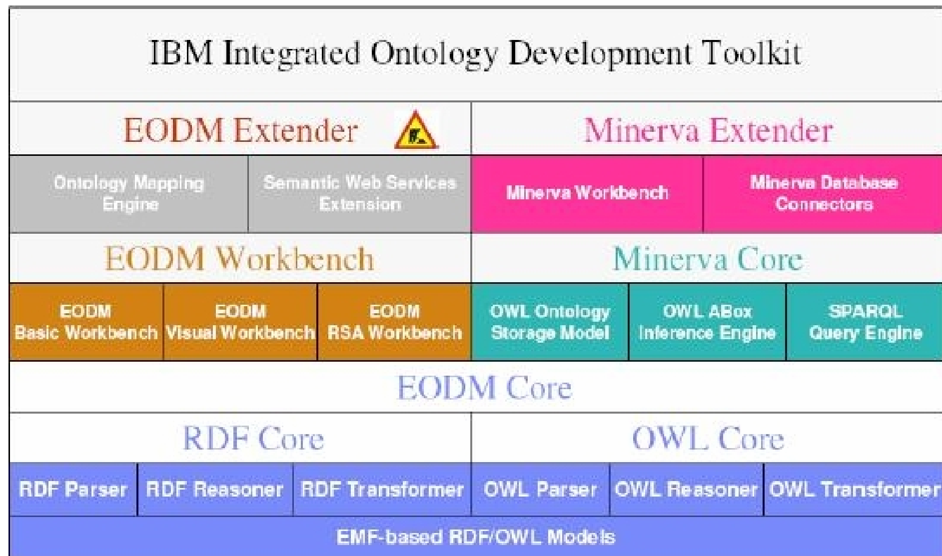

Figure 12: Architecture of IODT [LiMa2006]

Following are some features of this editor.

- .Eclipse-based Tool: EODM workbench is an Eclipse plugin that can integrate with any other plugins.
- OWL Support: EODM workbench now can support OWL-DL constructs and generate .owl files.
- UML-like Graphic Notion: In addition to traditional tree-based ontology visualization, EODM workbench provides UML-like graphic notion. Class, DatatypeProperty and ObjectProperty in OWL share the similar notion as Class, Attribute and Association in UML. Detailed properties of OWL constructs are shown in the Property view.
- Multiple Views : In EODM workbench, one ontology can have multiple views to support visualization of huge ontology. These views are independent, and users can decide to delete something from ontology permanently, or from view only. Multiple views will be synchronized automatically.
- Class/Property Hierarchy: EODM workbench keeps the tree-based editor for class and property and provides some shortcut to create subClassOf and subPropertyOf in the hierarchy. Users can drag class and property in the tree to the canvas in the view.

2006–2007 © Copyright lies with the respective authors and their institutions.

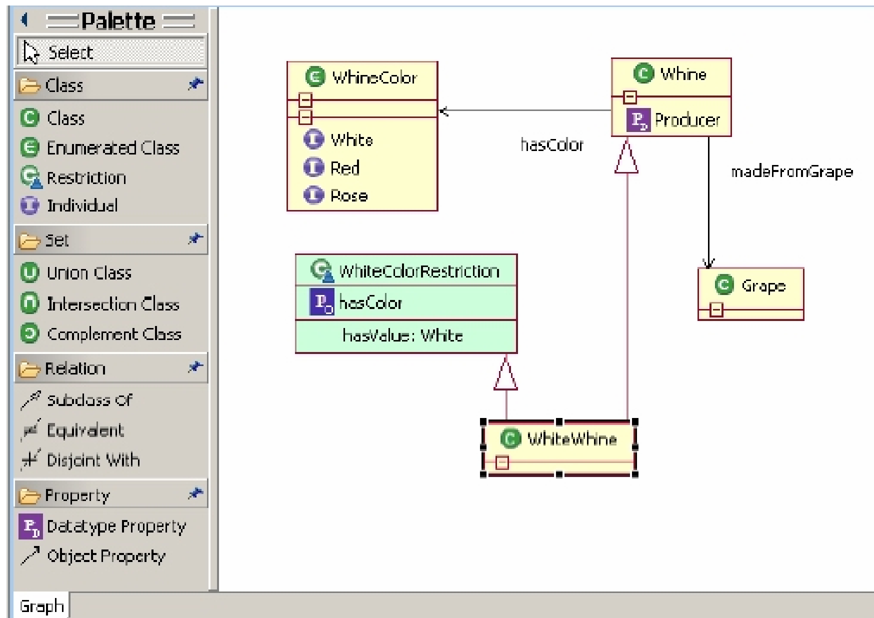

Figure 13: IODT graphical editor

3.2.1.7 SWOOP v2.3 beta 3:

SWOOP is a hypermedia-based featherweight OWL ontology editor, created by mindswap. Design and usage of Web-browser make this editor be more effective (in terms of acceptance and use) for the average web user by presenting a simpler, consistent and familiar framework for dealing with entities on the Semantic Web. Following is some detailed advantage of this editor. The figure below is the architecture model of SWOOP.

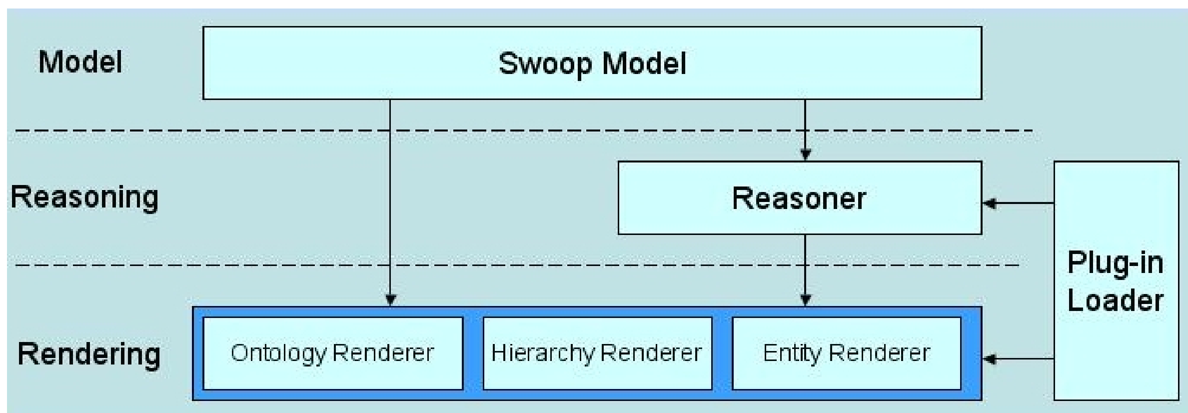

Figure 14: Swoop Architecture

- Web Browser like look & feel: An address bar where the URI of the ontology (or class/property/individual) can be entered directly for loading; hyperlink based navigation across ontological entities(address bar URL changes accordingly); history buttons for traversal and bookmarks that can be saved for later reference. The figure below shows the Web Browser like.

- Inline Editing: All ontology editing in SWOOP is done inline with the HTML renderer. using different color codes and font styles to emphasize ontology changes. Undo/redo options are provided with an ontology change log and a rollback option.

- Presentation syntax tabs: Concise format view; abstract syntax(from OWL API); RDF/XML ; Turtle/N3; plug-in architecture for new tabs.

- Multiple Ontology Support: Browsing, Mapping, Comparison.

- Built-in reasoner and Debug mode: By exposing the internal workflow of a Description Logic Tableaux reasoner (Pellet) in a meaningful and readable manner, explanations are provided to help users understand the cause for (and remove) inconsistencies detected in ontologies.

2006–2007 © Copyright lies with the respective authors and their institutions.

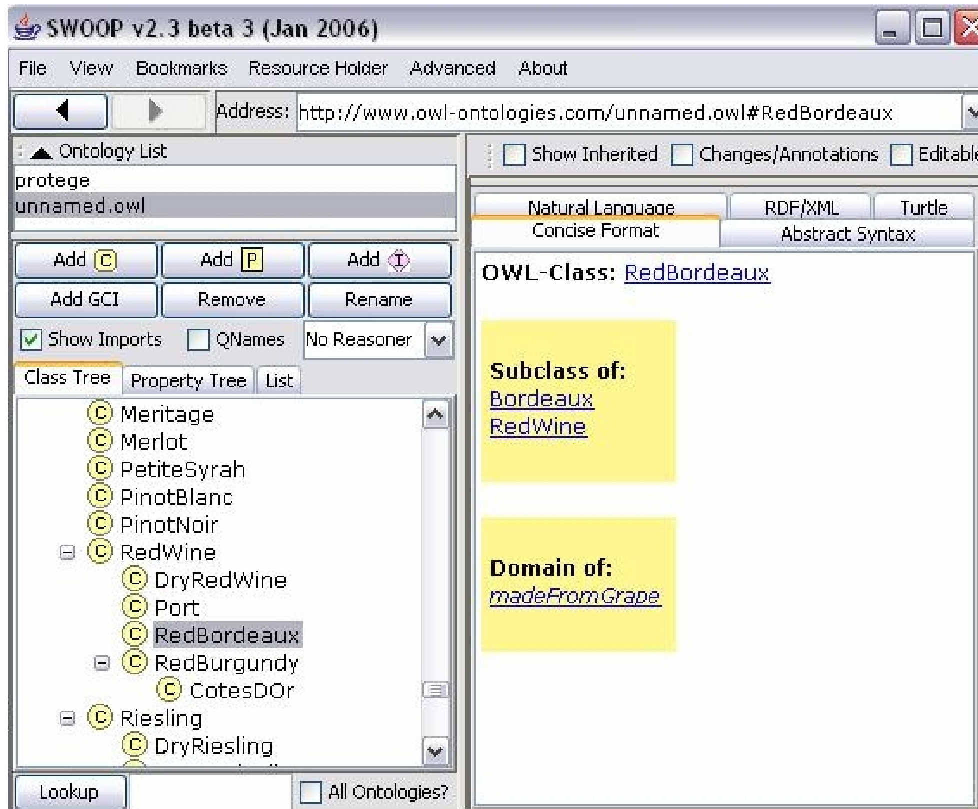

Figure 15: SWOOP GUI

3.2.2 Summary and Conclusions

By describing in detail about the features of the OWL editors mentioned above we can get some useful ideas about how the future OWL editor would look like. The OWL-DL editor would be better used when it contains the following features.

- Eclipse-based : easily integrate with many plugins
- Visualization: edited graphically. This is helpful feature used by the novice.
- Multiple View: views can be quickly changed.
- Reasoner built-in: easy and fast checking of consistency and debugging especially for the beginner.
- Menus and wizards: can help user build the ontology easily.

Concerning OWL-Editors, some tables have been made to compare the editors mentioned above. Hence the efficiency and usability of the OWL editors can be shown more clearly.

Table 1 shows some general information about the editors. For instance, how the editors look like, if they support OWL DL files directly or not and which system they are based on.

Table 1: OWL Editors: general information

	Protégé 3.2 Beta	Altova SemanticWorks 2006	TopBraid	SemTalk 2	IDOT	SWOOP v2.2.1
View	Form Text	Form Text Graph	Form Text Graph(Mix)	Graph	UML-like Graph	Web Browser-like
OWL DL file Support	Direct	Direct	Direct	OWL.vst	Direct	Direct
Based on	Java	.Net	Eclipse Jena Pellet	Microsoft Visio Microsoft.NET Framswok	Eclipse EODM Minirva	Java

Table 2 shows the lists of import and export formats of each editor, which provide a good overview of their interoperability choices.

Table 2: OWL Editors: Interoperability

	Protégé 3.2 Beta	Altova SemanticWorks 2006	TopBraid	SemTalk 2	IDOT	SWOOP v2.2.1
Import	RDF(S) OWL	XML RDF(S) OWL	XML RDF(S) OWL	HTML/XML RDF(S) OWL	XML RDF(S) OWL	RDF OWL DAML
Export	XML RDF(S) OWL	XML RDF(S) OWL	XML RDF(S) OWL	HTML/XML RDF(S) OWL	XML RDF(S) OWL	RDF OWL DAML

One of the important features of ontologies using OWL-DL is that they can be processed by a reasoner. **Table 3** shows the most relevant information about Reasoners. One of the main services by a reasoner is the subsumption test, meaning whether or not one class is a subclass of another class. A similar service is the automatic classification by determining for all instances its classes and for all classes its super classes. Another standard service that is offered by reasoners is “Consistency Checking”. Based on the description (conditions) of a class the reasoner can check whether or not it is possible for the class to have any instances.

2006–2007 © Copyright lies with the respective authors and their institutions.

Table 3 also shows the list of inference engines that can be used with the editor. Some inference engines are integrated in the editors. It makes the editor easily be used.

Table 3: OWL Editors: Reasoning

	Protégé 3.2 Beta	Altova SemanticWorks 2006	TopBraid	SemTalk 2	IODT	SWOOP v2.2.1
Consistency Checking	Yes	Yes	Yes	Yes	Yes	Yes
Automatic classification	Yes	No	Yes	No	No	No
Inference engine	Attached via DIG	None (simple checks)	Pellet (built-in)	RACER FACT (ect.)	Racer(built-in) Pellet(built-in) TBox(built-in)	Pellet(built-in)

Not all OWL primitives can be used in OWL Lite, such as `owl:disjointWith`, `owl:oneOf`, `owl:unionOf`, `owl:complementOf`, but they can be used in OWL DL. Consequently, it is useful to make a table to know if the editors support OWL DL by trying the primitives mentioned above in these editors**. Table 4** shows the results.

Table 4: OWL Editors: OWL-DL KR primitives

	Protégé 3.2 Beta	Altova SemanticWorks 2006	TopBraid	SemTalk 2	IDOT	SWOOP v2.2.1
disjointWith	Yes	Yes	Yes	No	No	Yes
unionOf	Yes	Yes	Yes	Yes	No	Yes
complementOf	Yes	Yes	Yes	Yes	No	Yes
oneOf	Yes	Yes	Yes	Yes	Yes	Yes

4. General Architecture

The general architecture of NeOn is structured into three layers (see Figure below). The layering is done according to increasing abstraction together with the data- and process flow between the components. This results in the following layers:

- Infrastructure services: this layer contains the basic services required by most ontology applications.
- Engineering components: this middle layer contains the main ontology engineering functionality realized on the infrastructure services. They are differentiated between tightly coupled components and loosely coupled services. Additionally interfaces for core engineering components are defined, but it is also possible to realize engineering components with new specific ontology functionality.
- GUI components: user front-ends are possible for the engineering components but also directly for infrastructure services. There are also a predefined set of core GUI components.

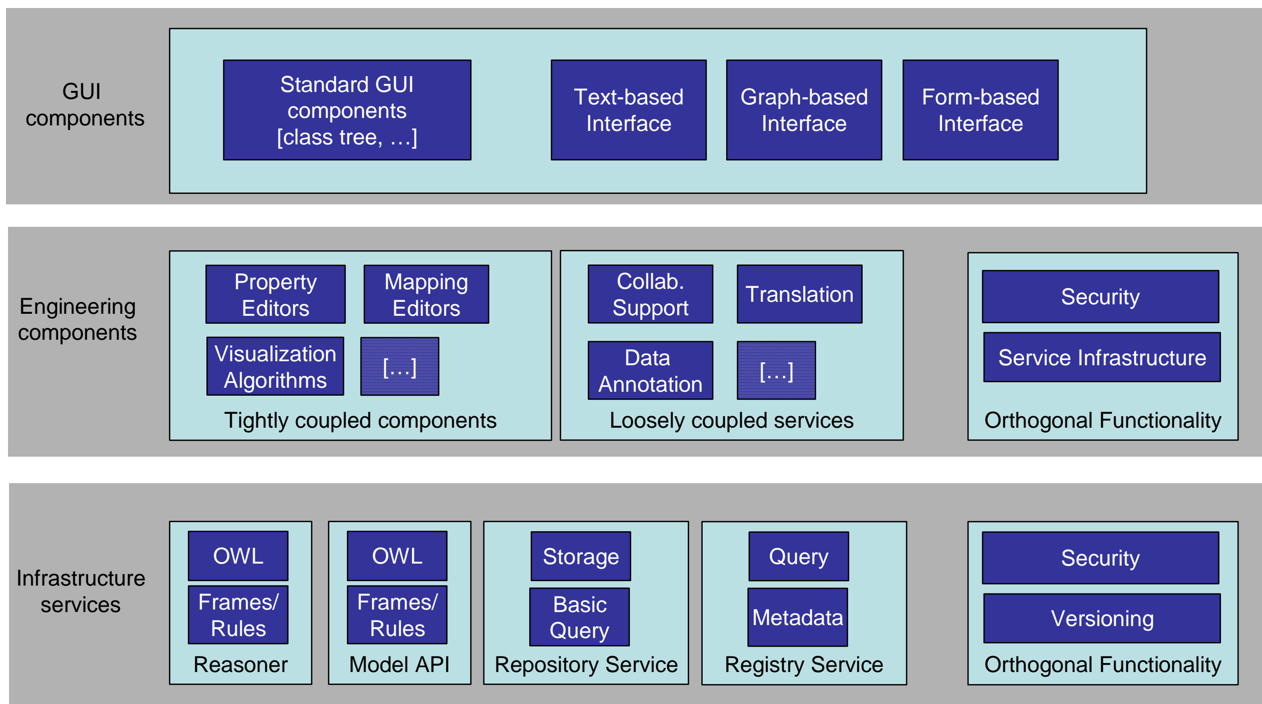

Figure 16: NeOn Toolkit Architecture

4.1 Core concepts

4.1.1 Layering of architectural components

We discuss in the following the chosen layering approach by distinguishing between:

- Complete abstraction layers

2006–2007 © Copyright lies with the respective authors and their institutions.

A complete abstraction layer means that all components of higher level use only next lower components.

- Layer of service usage

  The distinction between infrastructure services and engineering services is not a very strict one. It is just based on the categorization into infrastructure and lower level data-oriented services.

  Thus the layers indicate control- and data flow in addition to abstraction. A single operation on a higher level can result in several "low level operation". If a user for example triggers a delete operation using a particular control (such as a tree control for class hierarchies), this might result in several operations on the language model layer (and even more operations on the FOL layer). Subordinate functionalities such as session management or security support might also be realized on several layers, using different levels of abstraction.

A complete abstraction layer is too inflexible for the different characteristics of the services in an ontology engineering environment. A layer of service usage is however useful as it allows to distinguish between fix basic infrastructure services and pluggable engineering components.

4.1.1.1 Functionality vs. non-functional requirements

First of all there is the „natural" functionality of a layer, such as storing and managing first order logic statements or language-bound statements. However, looking at layers independently can lead to typical bottlenecks. A component to edit models (using a text control or a form) might require a function to support automatic completion proposals for typed statements. The computation of completion proposals might not be seen as a "natural" function of the language layer responsible for the language primitives (which form the base for the completion). But realizing this functionality on a higher level can cause serious performance issues. The same holds for certain refactoring functions. If the top level component depending on this functionality requires a particular performance to guarantee some degree of interactivity, then there are basically two options: 1) bridging the layers (see below) and accessing the low-level storage directly at any level that is appropriate to realize the functionality or 2) designing the layers and interfaces in such a way that the core functionality is realized at the suitable layer and then encapsulated through several layers. The first option is difficult to realize and makes some basic advantages of the layered approach obsolete. Other options (such as caching large amounts of data lower levels are responsible for) are also in conflict with the layers.

Another example where the question of where to realize certain features comes up is the security issue. Basically security features could be realized on one of the higher levels such as the control level. However, this implies that lower levels are completely encapsulated and security mechanisms can not be bypassed bridging the layers. This is not realistic for data sources that play a role outside of the core layers.

- Layer Bridging

As described above, layer bridging is closely related to the question of functionalities and other requirements. For each layer it has to be decided, if layer-bridging is possible and from where it is possible.

- Data Source Abstraction

The abstraction of data sources is a core aspect of the architecture, as the layered design on top of a repository layer indicates. However, data source abstraction can also occur at particular points of the architecture, such as model import functions or database schema import. There are different approaches of data source abstraction based on different goals. The eclipse project targets data source abstraction from an engineering perspective, emphasizing the convenient management of

metadata. Application servers or containers provide data source abstraction (often focusing on relational databases based on OR-mapping) for runtime use. The OGSA-DAI framework emphasizes the distribution of data sources and provides a (stateful) service layer with security mechanisms, etc. It's still to be evaluated, which approach might be relevant for NeOn requirements regarding data source abstraction.

4.1.2 Service interfaces

4.1.2.1 Service-oriented vs. Object-oriented

The service-oriented approach is one major driving paradigm of the NeOn architecture. This holds especially for the communication of components across the different layers. The object-oriented approach plays a role on a more fine-grained level, e.g. regarding internal control structures of pluggable components.

However, service oriented does not imply the actual kind of service (e.g. web service). It basically allows for a loose coupling of components and additionally provides a base for distributed environments.

The following code snippets illustrate invocation aspect of both approaches

Object-oriented:

Rule.getHead()

Service-oriented:

getHead(String ruleID)

A major difference is however also the granularity of the functionality. Object-oriented methods provides usually very fine grain functionality like the one above. Opposite to this services provide very coarse-grain functionality

At the core layer of (distributed) components for knowledge- and ontology engineering tasks such as alignment service-oriented coupling occurs at the following points:

- integration of distributed components based on the service interface they expose;
- deploy-function for pluggable components.

4.1.2.2 Transport layer

Not only due to networked ontologies a lot of distributed components occur in the NeOn architecture. Therefore the question of the appropriate protocols in the transport layer arises.

- Configurable transport layer

  As mentioned before, the service-oriented approach as such does not imply the actual transport mechanism. Frameworks such as Spring support a configurable transport layer. This enables users to specify the transport layer implementation (RMI, JAX-RPC,…) via configuration file.
- Transport layer according to SOAP protocol

  The standards for web services offer also quite a large independence of the transport layer e.g.: http and soap binding. As they are increasingly accepted this independence offers enough flexibility.

2006–2007 © Copyright lies with the respective authors and their institutions.

Additionally by agreeing on SOAP as the common service protocol it is also possible to add orthogonal functionality in a consistent way. An important example is security support via the WS-Security standard.

Thus the NeOn architecture will use the SOAP protocol as the preferred transport protocol for all remote components.

4.2 Development architecture

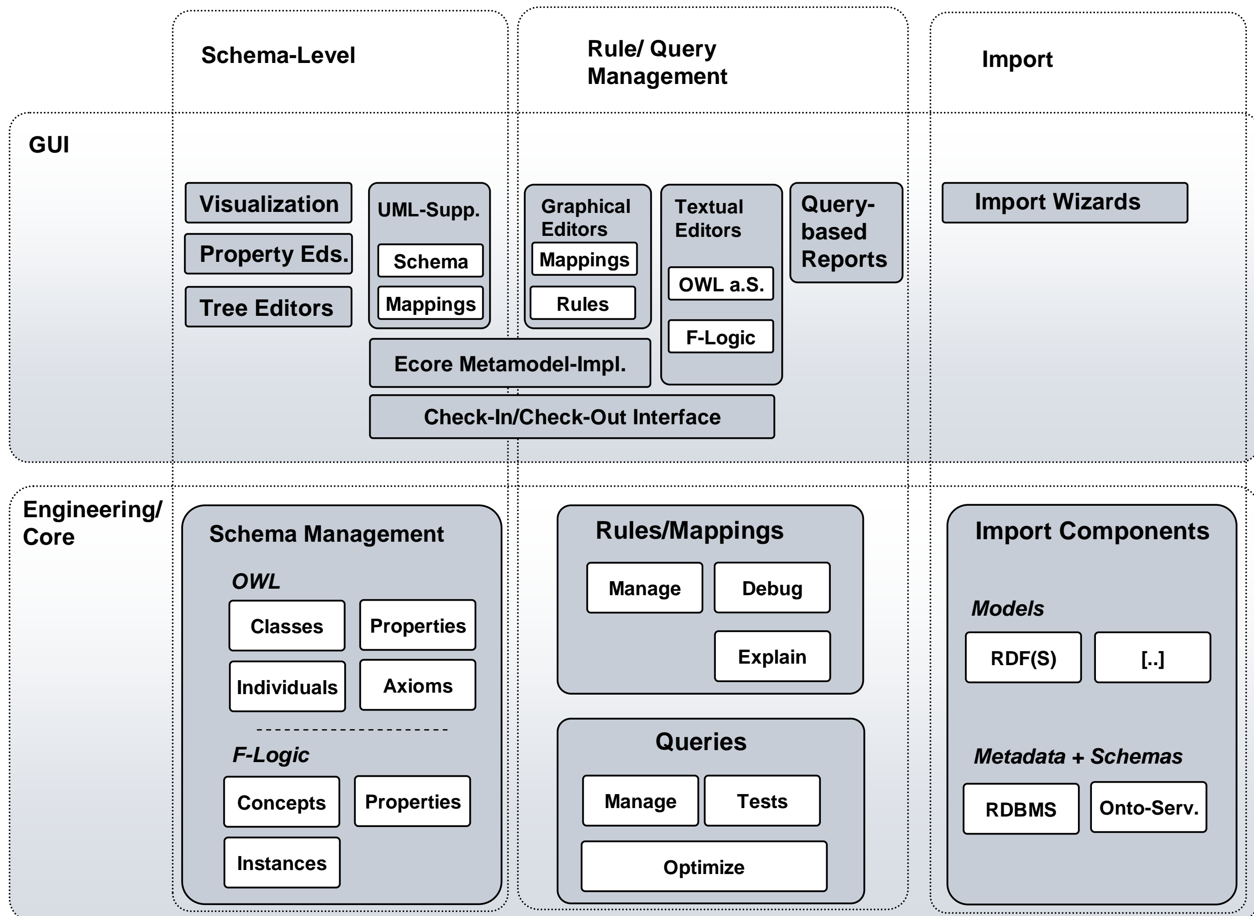

Figure 17: Core Components

The focus of Neon as an Ontology engineering platform is on the development architecture. This is shown in form of the core components in above figure, where the engineering core layer as well as the GUI-layer is illustrated.

Typical core components support management of ontology resources and access to reasoning services (e.g. for query testing). Resources might be created or imported and then edited. The editing of resources might be supported based on different editing paradigms, such as textual editing or graph-based editing. This is handled on the GUI-layer.

The usual form of implementation for a GUI component is based on the eclipse framework, using JFace/SWT.

The core layer provides access to the ontology-data model. Plugin developers will interact with ontology-objects such as OWLEntity, OWLClass, OWLAxiom, etc. (see section 4). It will in addition to the API provide control structures that support typical editor features. This includes for example the graphical editing based on the metamodel implementation (UML-support) or textual editors. The problem with this kind of editing support is the synchronization with the core data model. To support this we introduce a thin layer above the language API that controls “bulk” editing. Plugins might check out an ontology and check it in again without intermediate synchronization.

The core layer will be reflected by a datamodel-plugin similar to the current plugin of OntoStudio. However, there will be a better “visibility” of the underlying API, which means that plugin developers will have access to elements like OWLClass (see section 4.). It will also provide additional control structures for typical editing functionalities (see subsequent sections).

4.2.1 Use Cases

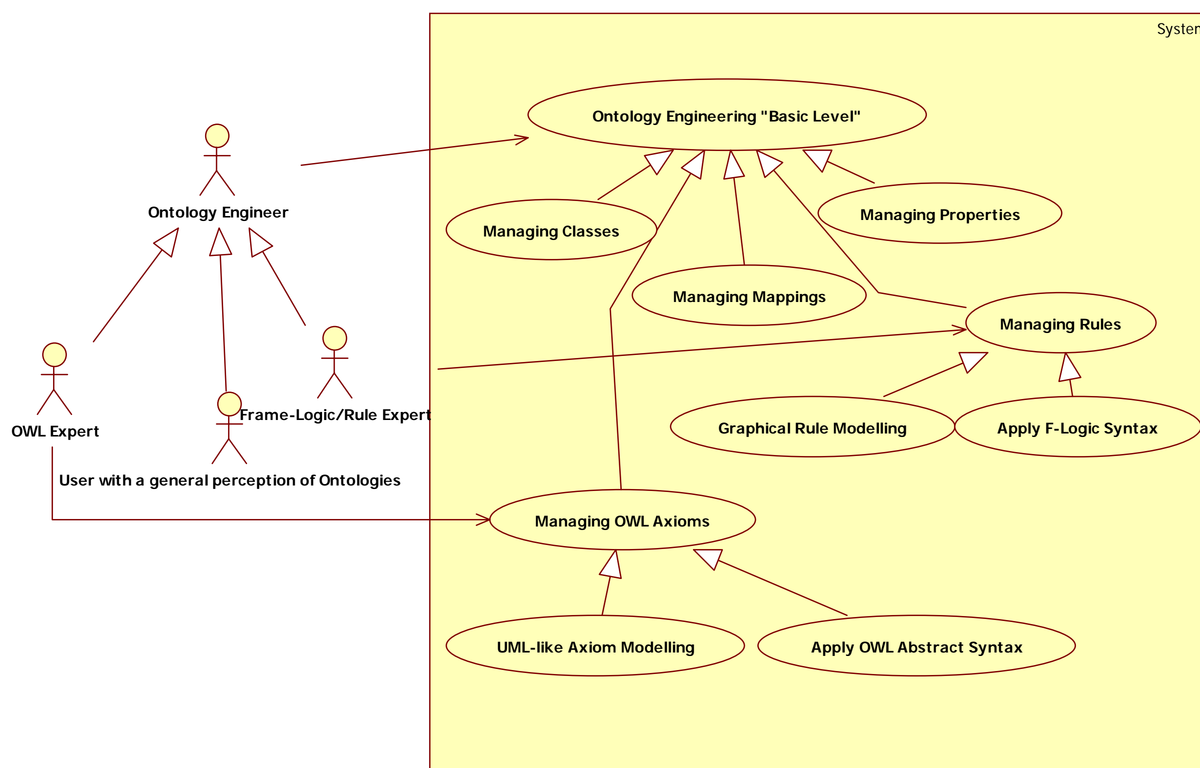

Figure 18: Ontology Engineering Use-Cases (Examples)

There are different use cases for the engineering layer and GUI components. The above figure shows an example for use-cases on the end-user level. Ontology engineers with different skills are expected to access different kinds of engineering functionalities.

The figure below shows a use-case where a user extends the toolkit itself. While much of this use-case is determined by the eclipse framework (see section 3.1), the toolkit provides additional mechanisms for extensions, such as plugin-development wizards. In figure below this is illustrated for the case of an import-plugin. Developing import-plugins allows to support additional formats (but not their formal semantics!). An import-plugin-wizard would generate the “skeleton” for the plugin implementation. It would be the task of the user to implement the logical mapping from the supported format onto either F-Logic or OWL and implement the user-interface for import-specific

2006–2007 © Copyright lies with the respective authors and their institutions.

metadata. The workflow as well as the integration into an existing import menu would be determined by extension points. The code generated by the wizard implements the functionality for those extension points. Developers then would not have to care about questions like how to plug into a certain menu (such as the choice of import formats).

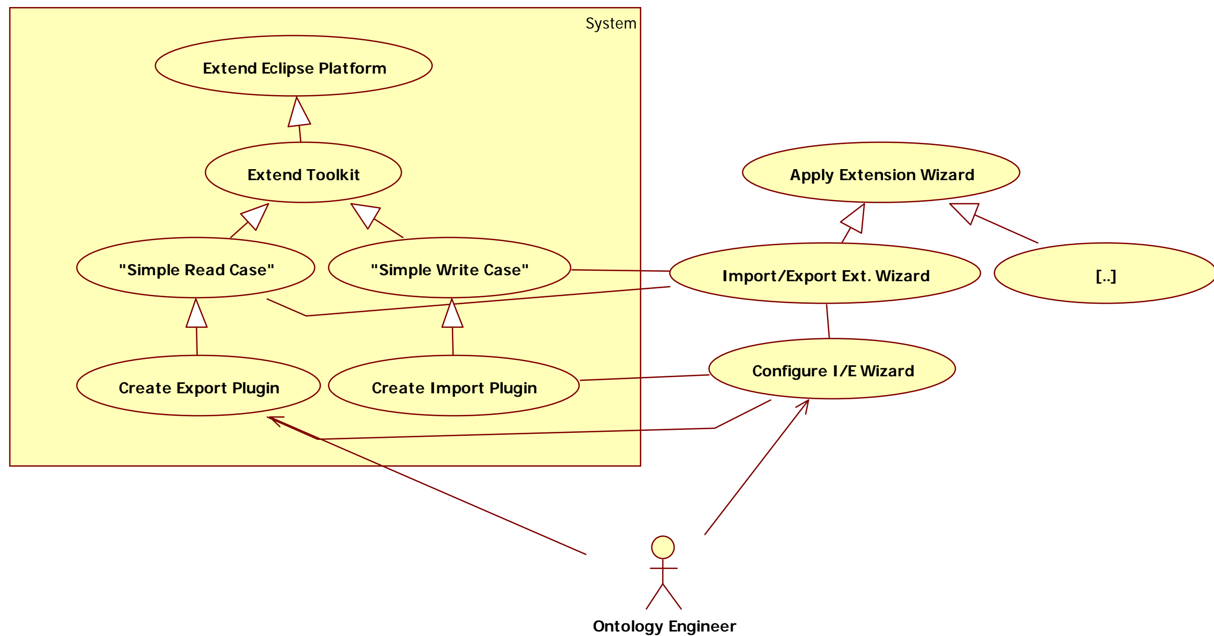

Figure 19: Extension Use-Cases (Examples)

4.2.2 Layer 3: GUI components

The usual implementation of a GUI component would be a separate eclipse-plugin that is the counterpart to another plugin containing the engineering functions. Basic GUI components will be part of a minimal configuration of the toolkit. This includes typical property editors for the management of language elements like classes, properties, axioms.

4.2.3 Layer 2: Engineering components

The engineering components are the main source of functionalities that end-users typically make use of. An engineering component consists of one or more plugins. The basic engineering operations (managing elements of the ontology language) are supported through a core plugin. Other plugins would typically fall into certain categories.

Categorization of Plugins

The following categories help to understand in which way the engineering level of the toolkit might be extended. The differentiation between instance level and schema level is based on the observation that there are fundamental differences in how the respective elements are handled and what roles they play.

Figure 20 shows a categorization of plugins along the way the available ontologies are accessed. An export plugin for example would require access to the complete model in order to serialize it. A plugin for a textual editor (supporting a certain syntax) would require “bulk read” as well as “bulk write” access, except if the editor is applied to certain elements (single axioms, F-Logic rules, SWRL-rules, …).

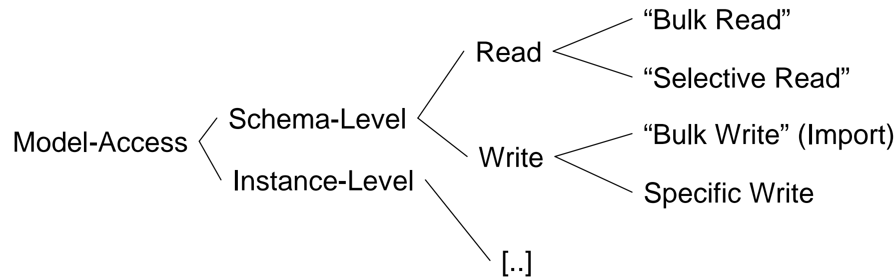

Figure 20: Plugin-Categorization: Model-Access

Editing plugins requiring synchronization can be categorized as illustrated in figure below. The toolkit will support the “large model at once” option through editors that are in line with a repository paradigm. This includes for example tree views, that fetch elements in a very economic way. Single entities such as classes or properties can be manipulated with typical property editors.

The medium level is likely to be supported only in an unsynchronized way as described in section 3.2..

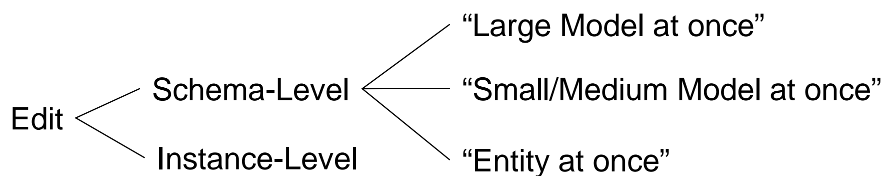

Figure 21: Schema Editing Plugins

The Figure below shows the context that a certain plugin might require. A context is a state in which a certain element or several elements are selected. The toolkit will support several contexts for different elements (like ontologies or single classes) through the respective extension points.

2006–2007 © Copyright lies with the respective authors and their institutions.

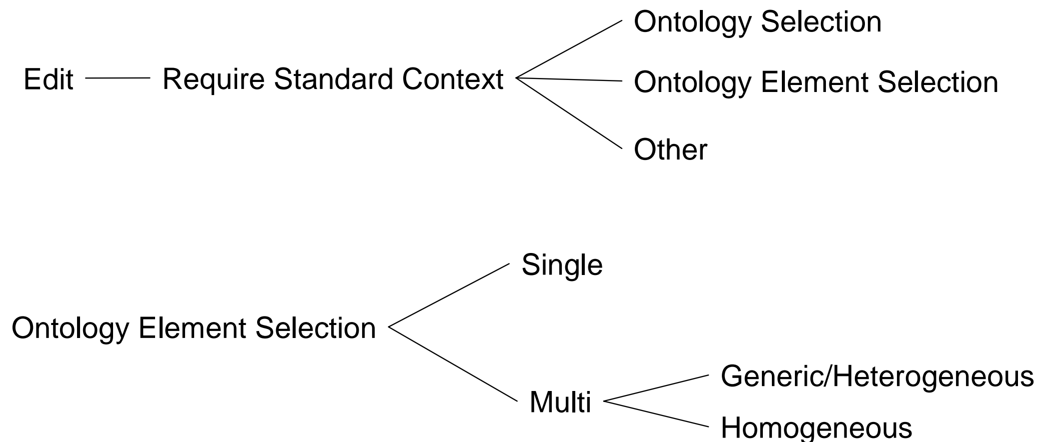

Figure 22: Selection Context

Export/Import Plugins

For export/import plugins the following cases are supported:

- Import to: OWL(DL)
- Export from OWL(DL)
- Import to F-Logic
- Export from F-Logic
- Menu entries and Wizards

Edit-Controls

Edit-controls include the property-editors mentioned above, tree-control based editors for class hierarchies and refactoring features. Graph-based editors and textual editors will be an alternative environment for complete models and for single elements (rules, axioms).

Visualizing/Browsing

- Visualize OWL-ontology
  - Provide layout-algorithm
  - Provide
- Visualize Rule-Graph
- Visualize Frame-ontology

4.2.3.1 Coupling of components

Concerning the coupling to the toolkit we distinguish between two main categories:

Tightly coupled component

The characteristics of tightly coupled interfaces are

- Highly interactive components
- Fine grain component
- Locally used components
- Using NeOn repository for persistence states

Tightly coupled components are realized as conventional Eclipse plugins. Examples are Mapping editors or Ontology browsers. The NeOn toolkit will use specialized Eclipse features as the realization mechanism for tightly coupled components. This has been explained along the example of an import plugin in the previous sections.

A typical example of a tightly coupled component would be a plugin with a rich graphical interface where frequent user actions invoke process on the infrastructure layer. A tightly coupled component directly interacts with the infrastructure layer without any transport layer in between. Tightly coupled components would include for example:

- property editors;
- textual editors;
- graph-based editors.

Loosely coupled components

The characteristics of loosely coupled interfaces are

- Non interactive components
- large grain components
- remotely used components
- eventually having their own (additional) repositories

Thus the loosely coupling allows to use functionality in the toolkit, which was independently developed or cannot be easily deployed into the toolkit environment. Examples are specialized reasoning services or ontology annotation tools for text, which require a specialized infrastructure.

Loosely coupled components as web services

In the NeOn toolkit the loosely coupled components have to be available as web services. This requires the realisation of a usually thin web service layer on top of the component. If not already available the web service layer should be realized in the hosting environment of the component in order to avoid too many protocol indirections.

For many standard infrastructures tools are available to generate such a web service layer e.g.

For Java components: Axis , …

For .NET components: .NET framework, …

2006–2007 © Copyright lies with the respective authors and their institutions.

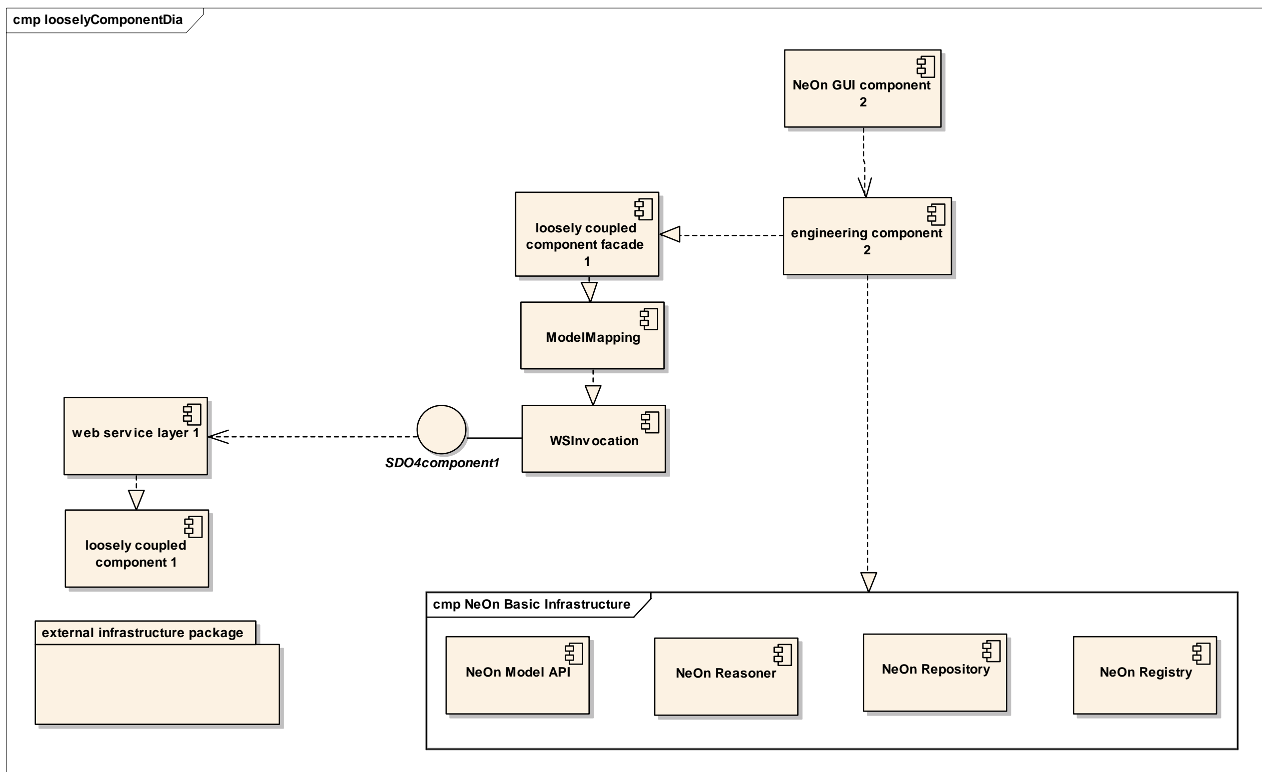

Figure 23: Loosely coupled engineering components

Interfaces for loosely components

Loosely coupled NeOn components should also be easily integrated with the other components of the NeOn toolkit. Therefore they should use the NeOn Ontology model in their interfaces. For web services this means an XML representation of the models. For the currently four parts of the NeOn Ontology model they are defined in the following, which are the same representations in the web service versions of the NeOn APIs:

- NeOn OWL model: direct XML representation of OWL1.1 standard [Grau2006] (see also XML schema http://www.w3.org/Submission/owl11-xml_syntax/schema/owl1.1.xsd).
- NeOn rule model: XML representation of W3C RIF group [Ginsberg2006]. As this is in development the OXML format of Ontoprise is currently used.
- NeOn mapping model (adoption of the XML representation of the NeOn rule model)
- NeOn modular ontologies (adoption of the XML representation of the NeOn rule model)

Loosely coupled components as web services

The NeOn toolkit will use specialized web services as the realization mechanism for loosely coupled components.

Integration of loosely coupled components into the NeOn toolkit

For a loosely coupled component mechanisms are needed to make the functionality of a web service available as an engineering component. This requires components for:

- Invocation of the remote web service in Eclipse development environment
- Transformation of web service XML data to java objects
- Mapping of the ontology model of the foreign web service to the NeOn ontology model
- Presentation of the functionality as an Eclipse plugin-based engineering component

SDO is a suitable mechanism to transform the web service XML data to java objects. It allows both a static binding for the fixed data of the web service to be invoked. But it is extensible via a dynamic binding which is required for the handling of open content like annotations.

Most of the components needed for this loose coupling can be supported by generating templates for such a plugin based on the WSDL description of the remote web service.

Usage scenarios of loosely coupled components

- Existing web services
  - Interface adoption to NeOn model only during invocation into NeOn toolkit
  - Preferable usage of SDO to realize necessary adapters in NeOn toolkit
- Existing remote components
  - Interface adoption to NeOn model in web service layer on top of existing remote component
- Newly developed loosely coupled components
  - Component can directly use NeOn model in interfaces

4.3 Layer 1: Basic Infrastructure Services

There are the following approaches to define the functionality of the basic infrastructure services:

- Repository related functionality

  The basic infrastructure layer contains all repository related functionality, which is needed by the engineering components like storing and basic query mechanism.
- All functionality needed by engineering components

  A more flexible approach is to add to the basic infrastructure layer all the functionality which is usually needed to realize engineering components. It should be organized in always required functionality like repository and NeOn ontology model support and optional functionality.
  These basic infrastructure components are not the same as predefined engineering components, which are needed for an ontology engineering platform (like an ontology editor). However they are not needed to realize another ontology engineering component.

The NeOn architecture follows the last approach to include all basic functionality needed by engineering component as infrastructure services. Our analysis has shown that the following components are the most critical:

2006–2007 © Copyright lies with the respective authors and their institutions.

- NeOn Model API
- Reasoner
- Ontology Repository
- Ontology Registry

The interfaces of those components will be available as web services and as Java interfaces. Thus they can be used also in loosely coupled components. Via the web services also orthogonal functionality like security can be added in a generic way.

4.3.1 NeOn Reasoning Service

The NeOn reasoning service(s) are a core component of the toolkit on the infrastructure level. Reasoning support will be given as an internal service of the toolkit. This includes the reasoning capabilities of the KAON2 reasoner as well as the capabilities of the OntoBroker reasoner. They are described in the project deliverable D6.3.1.

4.3.2 Ontology Repository

The term repository often subsumes a wide variety of functionality. For the NeOn development architecture the functionality of a development repository for ontologies is essential. Another important aspect is the suitability of the repository for reasoning.

There are use cases where a simple repository functionality is sufficient. However for the overall NeOn goal of supporting large scale ontologies more sophisticated repository functionality is needed. In order to support a distributed environment adequately a good integration with the registry functionality is necessary. Reasoning on large ontologies additionally requires specific repository functionality for fast access to selected parts of many ontologies. Thus a common repository API has to be defined, which allows different implementations supporting the different characteristics of the specific requirements.

Besides different implementations of common functionality a lot of requirements also mean different functionality. Therefore we define in the NeOn architecture a repository API with a core functionality and certain optional functional components. Implementations can choose the appropriate functional level, which is adequate for their specific requirement characteristics.

4.3.2.1 Core Functionality of repository

The repository manages ontologies identified by a unique name given as an URI in a persistent store. They are organized in a hierarchical, directory-like structure of collections. The functionality is based on the WebDAV protocol.

The ontology repository manages directly all the artefacts needed for an ontology-based application. This consists of the following kind of data:

- Ontologies according to the NeOn ontology model. This includes the OWL and the rule model but also the mapping and module model extensions
- Other data in the following formats, where the differentiation is based on the mime-type.
    - XML
    - text
    - Binary

Important interfaces for the basic functionality are

- Connection
- Collection
- Ontology
- Other resources

Basic Retrieval

The basic functionality is to access complete ontologies in a persistent store. It requires to identify the repository container itself. This is done by an URL. There are two basic mechanisms to access the ontology:

- Direct access
- Navigation

The direct access allows to directly access the ontology by an URL, which specifies its location in the repository. Thus it is usually different to its unique name URI.

With the navigation it is possible to traverse the hierarchical structure of collections, where the ontologies are located. This includes navigation to parent, children, successor and predecessor.

Manipulation operations

The basic manipulation operations are:

- Creating ontology in a specified collection
- Updating ontology given by location
- Deleting ontology given by location

4.3.2.2 Optional repository functionality

Query and reasoning capabilities of repository service

For the tradeoff of easy realisation and powerful functionality the degree of query and reasoning capabilities for the repository service is essential. We distinguish between three main levels

1. Basic Retrieval

   The minimum query support for a repository service consists of the described direct access via ontology name and navigational operations. Additional simple retrieval based on a fixed set of properties for ontologies according to the OMV model like ontology name, author, creation-date are available via the ontology registry.

2. querying without inferencing

   A query language on the ontology objects allows arbitrary comparison predicates, Boolean expressions. The functionality is equivalent to query languages on other data models like SQL and XQuery. A candidate for a query language is SPARQL. The simpler functionality without inferencing can be realized by a mapping to a query language like XQuery. This query functionality does not only allow the selection of complete ontologies based on arbitrary qualifications on their content but allows also to extract arbitrary parts of ontologies.

3. reasoning in repository service

2006–2007 © Copyright lies with the respective authors and their institutions.

The most advanced functionality is to incorporate reasoning into the ontology repository service. That means queries on the ontologies will be evaluated also on derived knowledge. For the realisation a reasoner is required.

Versioning

Versioning is an optional functionality. We provide basic versioning support for ontologies, which will be extended by more advanced collaboration facilities in specific engineering components. The basic versioning support is based on DeltaV versioning, which is offered via the subversion protocol. The granularity is an ontology document. It includes checkin and checkout facilities.

The rationale for subversion compared to CVS is mainly

- subversion is based on general internet protocol http compared to specific protocol of CVS, which cannot be used across fire walls
- extensions of CVS functionality with versioning of collections
- the functionality of subversion is a superset of the CVS functionality

Security

Another optional functionality is the security support. As ontology are used to model all kind of data it is in many cases critical that the access to the ontologies can be controlled completely. This is especially true in open environments like the semantic web.

As the repository functionality is based on the WebDAV functionality the use of the access control protocol [Clemm2004] is a consequent choice. It is a separate component for the WebDAV protocol. It defines a quite powerful access protocol. The actors are user and groups. They can have privileges defined for all operations on any resource. These access control elements are grouped to access control list for each resource.

Transactions and multi-user capabilities

Multi-user capabilities are part of the versioning support. Transactions are not available at the interface level as the versioning support offers sufficient mechanisms for isolating the changes of concurrent users. An optional functionality is that the repository operations are atomic in the sense that their effect is either completely visible or not all. This is typically realised by transactions of DBMS. Nevertheless a simple file based realisation is also possible, which does has not that guarantee.

4.3.2.3 Realisations

The core and optional repository functionality is expected to be realized by different realisations. Examples are

4.3.2.4 File based

The simplicity of the core functionality allows a relatively easy realisation on files.

4.3.2.5 RDBMS based Repository

An RDBMS based realisation of the core functionality is also quite simple. In addition an RDBMS can support the non-inferencing queries without too much effort and efficiently. Also the atomicity of the repository operations can be guaranteed internally by transactions. It can also offer security.

4.3.2.6 RDF-based

Specific RDF repositories allow additionally also the support of inferencing queries.

4.3.2.7 XML-DBMS based Repository

An XML-DBMS based repository allows complete support of most optional repository features especially on the versioning functionality and the security functionality. It is also very suitable to support non-ontological resources especially in the XML representations. However the query capabilities will not support inferencing in order to allow a direct mapping to XQuery capabilities. With that configuration inferencing queries will be directly supported by the reasoner on a set of ontologies fetched from the repository.

For the NeOn toolkit such complete repository functionality will be realised based on the CentraSite repository.

4.3.2.8 Datastore

For the realisation of the repository services a datastore is necessary. This can be

- RDBMS
- XML database
- File system

The utilisation of the repository service with a specific datastore is usually hidden in the repository component. However it may be an advantage to exchange the data store. We distinguish the following approaches:

- Exchangeable datastore via predefined NeOn datastore interface
- Exchangeable relational datastore via JDBC interface
- Datastore component only internal to repository service

Via the predefined repository service interface it is already possible to exchange the repository layer including the datastore. An independent replacement of the datastore component for all possible repository components is hardly necessary and would impose too many constraints on a NeOn repository component. However for a certain category like RDBMS-based datastores the standard interface JDBC should allow the replacement to a large degree.

2006–2007 © Copyright lies with the respective authors and their institutions.

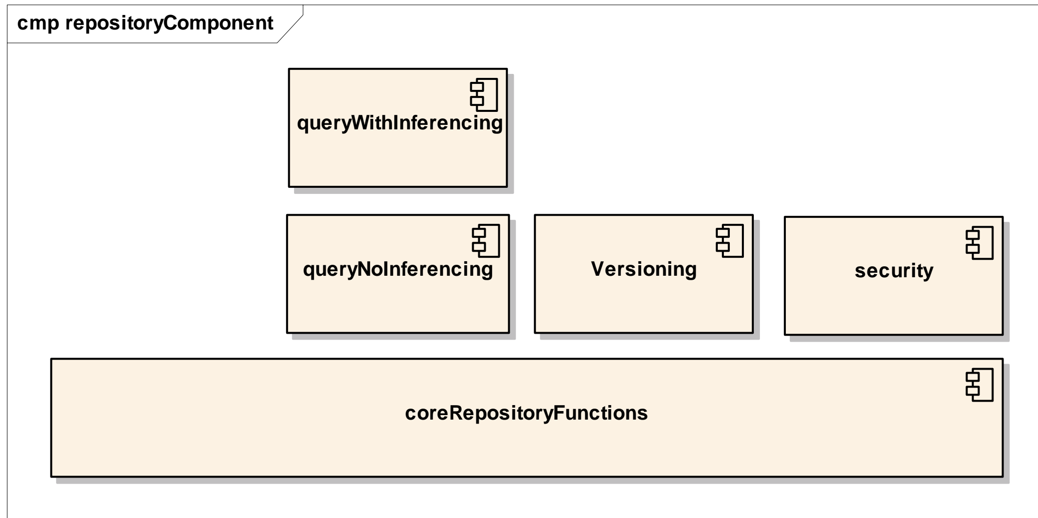

Figure 24: Repository functionality

4.3.3 Ontology Registry

With the increasing number of ontologies and their increased fragmentation into networked ontologies the need for an ontology registry is evident. This is not only true for the semantic web environment but also for large scale semantic applications in an enterprise environment.

The basic functionality of an ontology registry is based on an ontology meta model. For NeOn it will be based on the OMV ontology meta model. The ontology registry allows to register and query information about ontologies according to OMV. As this includes the location of an ontology in form of a directly accessible URL the ontology registry directly supports the management of many ontologies.

Besides this functionality for certain scenarios other critical characteristics are needed. Therefore we define a core registry API that allows several realisations with and without additional characteristics. Important additional characteristics are:

- Integration with repository: For complete governance in enterprise environments the registry has to be integrated with the repository.
- Integration with general purpose registry: Opposite to specialized ontology registry it is necessary for many real world usages of ontologies that the same registry can handle also other artefacts of the complete applications. Therefore the integration of the ontology repository functionality into a general purpose registry is needed.
- Federation: Several registries can act as a federated registry. This means mechanisms to synchronize the content of the federated registries. Another functionality are federated registry queries. They will be distributed to all registries of the federation. The results are sent back to the registry initiating the federation.

4.3.3.1 OMV specialized ebXML registry

For the basic infrastructure an API is needed to query, create and manipulate ontology meta information according to the OMV model. This API is defined as an ebXML registry API specialized for the OMV model. It exists in the form of a JAVA API and a web service interface.

Several OMV concepts are already quite well covered by the ebXML registry standard:

- General concepts: User, Organization
- General associations: uses, hasParent
- Version handling
- ID generation

The larger part of OMV is of course ontology specific. Thus a separate model in the ebXML registry is needed with associations to the general concepts like user and organizations.

2006–2007 © Copyright lies with the respective authors and their institutions.

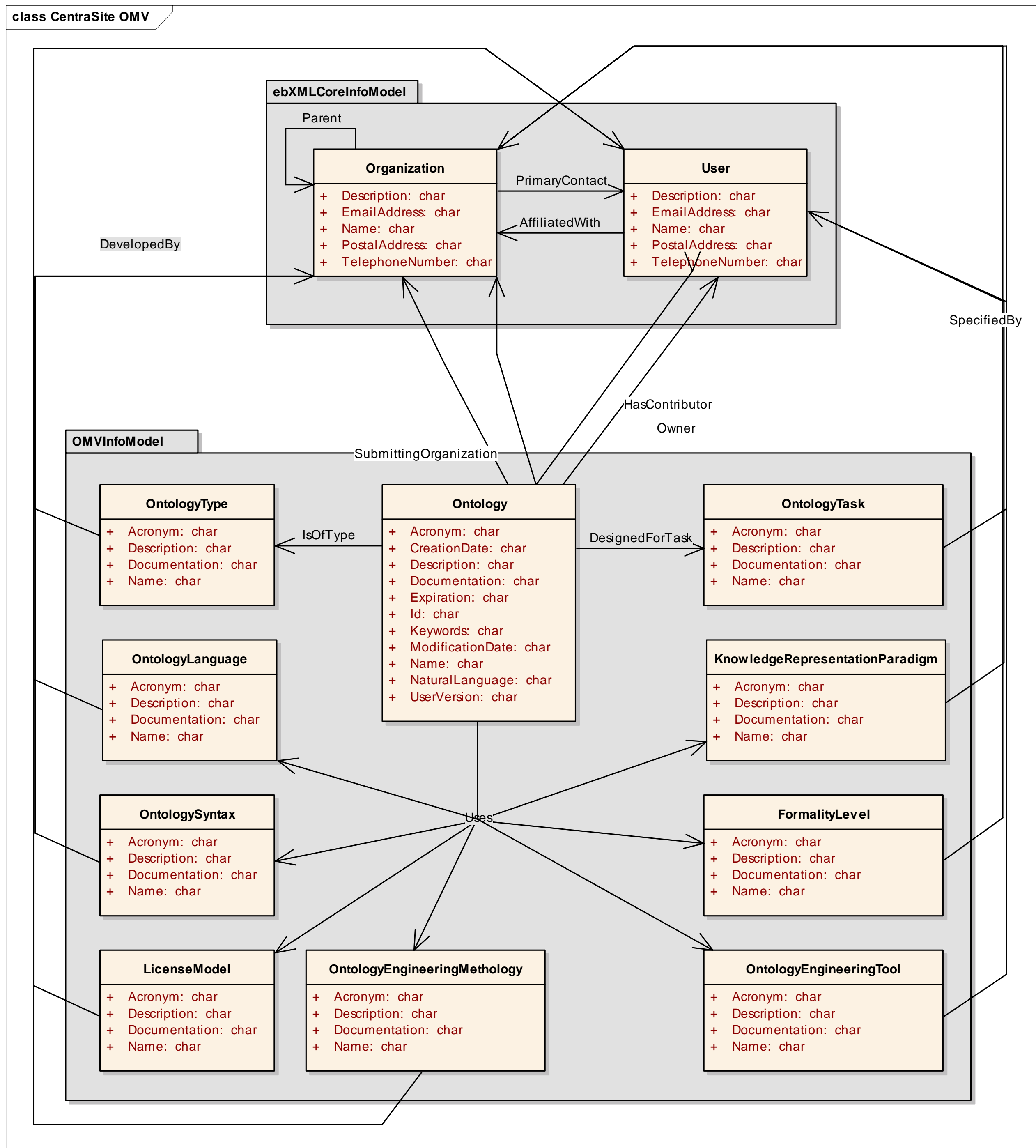

Figure 25: OMV Model

Based on this definition the standard ebXML APIs can be used. There are generic interfaces for arbitrary registry classes (subinterface of RegistryObject) differentiated into:

- QueryManager
- LifeCycle Manager

For the builtin classes like User and Organization a specialized Java API is provided:

In the following this is exemplified for the class Organization. For the OMV classes we provide in the same way specialized Query and Lifecycle operations.

That corresponds also to the Oyster Register API.

```
interface InfoModelLifecycleManager
public Organization createOrganization(InternationalString name)

public BulkResponse saveOrganizations(java.util.Collection
organizations)
public BulkResponse deprecateOrganizations(java.util.Collection
organizations)
public BulkResponse unDeprecateOrganizations(java.util.Collection
organizations)
public BulkResponse deleteOrganizations(java.util.Collection
organizationKeys)
…

interface Organization
public Key getKey()
public InternationalString getDescription()
public void setDescription(InternationalString description)
public InternationalString getName()
public void setName(InternationalString name)
public java.util.Collection getUsers()
public void addUser(User user)
public void removeUser(User user)
public java.util.Collection getChildOrganizations()
public void addChildOrganizations(java.util.Collection
organizations)
public void removeChildOrganizations(java.util.Collection
organizations)
…

interface InfoModelQueryManager
public RegistryObject getRegistryObject(java.lang.String id)
public BulkResponse
findOrganizations(java.util.Collection findQualifiers,

java.util.Collection namePatterns,

java.util.Collection classifications,

java.util.Collection specifications,
                                        java.util.Collection externalIdentifiers,
                          java.util.Collection externalLinks)
…


interface BulkResponse
public java.util.Collection getCollection()
```

2006–2007 © Copyright lies with the respective authors and their institutions.

```
public java.util.Collection getExceptions()
```

Additionally there will be a web service API for the same functionality. The ebXML registry services provide currently only a generic web service API, which is the same for all registry classes. There will be a specialized NeOn Ontology registry web service API, which operates on the OMV model.

Any registry interface where the OMV model is fixed will be similar to the specialized ebXML registry. This has the additional advantage of compatibility with other registry data. If it is realized in a general registry (like CentraSite) a lot of other registry functionality will be available on the registered ontologies: like impact analysis or standard UDDI functionality. As it has no real drawback compared to a proprietary API the NeOn registry API will be a specialized ebXML registry for the OMV model.

Reasoning on the OMV model

For a basic Ontology registry reasoning is not required. However the OMV model can be represented as an OWL/DL ontology. It is possible to export the content of the NeOn ontology registry as OWL instances. If reasoning on the ontology metadata is required an ontology with instances can be constructed with this functionality.

5. Language API

The NeOn language API is the core ontology interface of the NeOn infrastructure. It is the main access point for the basic ontology-related operations such as reading, creating and manipulating models (see figure below). The API is meant as a representation of the underlying languages encapsulating the details of interpretation, persistence, etc.

The NeOn API represents the lowest layer of abstraction for language-related operations applications or users should interact with.

This means that any of the use-cases displayed in figure below should be covered via the language API or an encapsulating API, never through elements "behind" the API. This is to ensure a clean layering and avoid problems with consistency, event-handling, etc.

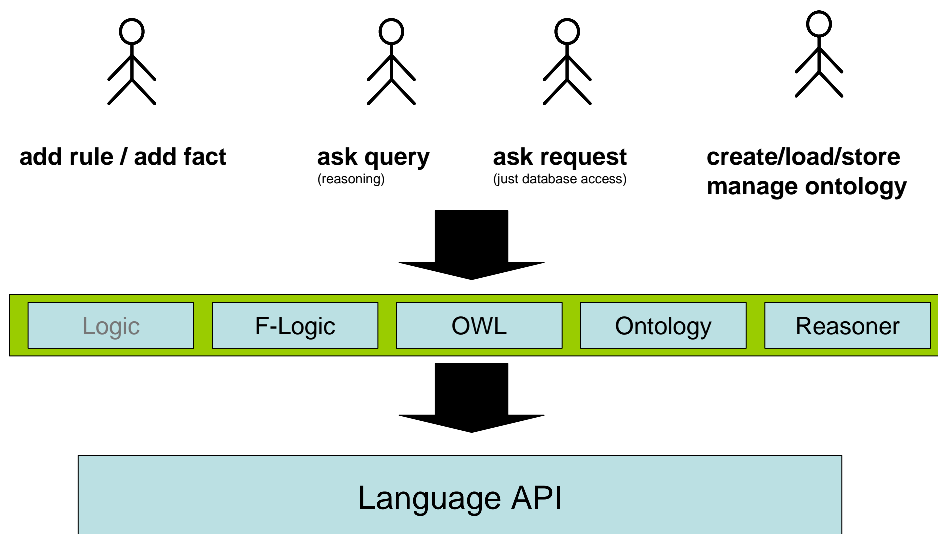

Figure 26: API Use-Cases

The base of this API is the KAON-2 API [Motik2006a], which has been brought together with the OntoBroker API.

Design

The API is meant to reflect the formal semantics of OWL and F-Logic based on a First-Order-Logic (FOL) layer. It allows applications and users interoperate with elements of the language (such as "OWLClass"). At the same time the API supports very flexible requests supporting efficient usage scenarios. The API allows for example to avoid certain inefficient filtering operations above the level of requests. This reduces the "degree of object-orientation" but supports scalable and efficient request-based applications such as the ontology-level of the NeOn toolkit.

While the API is a hybrid API supporting two languages it does not and is not meant to resolve the conceptual mismatch between different formal semantics of the languages. It is the base for hybrid applications and allows to harmonize infrastructure components (such as the fact base, certain inferencing components, the builtin-API, etc.). While the API as a common access layer is available, the harmonization of the API-implementations is still in progress (see below).

2006–2007 © Copyright lies with the respective authors and their institutions.

Implementations

As mentioned above, the API is implemented by two reasoners and ontology servers with different capabilities and different fields of applications. While both support different language-subsets and semantics, they both offer basic components with similar or identical roles, such as extensional databases, parsers, rewriters, built-ins, etc.

On a long-term range the goal is harmonize the internal interfaces of the components and achieve a modular implementation (see also section 4.1.).

Encapsulation

The API is a general ontology-API rather than a pure “editor API” though it supports functionalities typically required by an editor such as renaming operations. In the current design of the toolkit we plan a thin layer above the language API in form of a central datamodel plugin. The latter is the single point of access for plugins creating or manipulating ontologies.

Relation to Metamodel

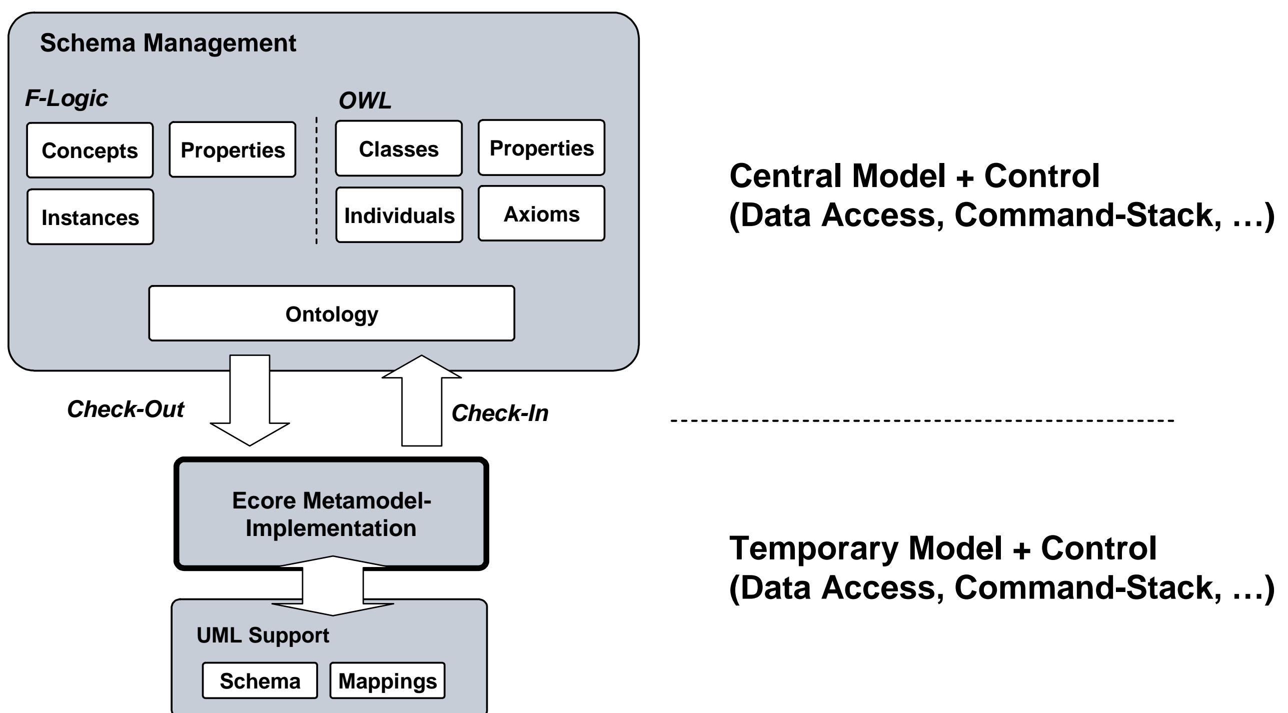

Figure 27: Relation between generated Metamodel implementation and Language API

The API largely reflects the NeOn model with the exception of elements that are currently not officially elements or attributes of the languages F-Logic and OWL (i.e. mappings). Work package 1 together with work package 6 will provide a generated 1:1 implementation of the NeOn-Metamodels. The latter will be mapped onto the API described in this section (work in progress). This does however will not happen in form of a layering but in form of a translation, supporting data-exchange in a batch-mode.

5.1 Role of the API

The API supports the management of ontologies as well as reasoning on ontologies. It thus supports engineering environments as well as runtime servers.

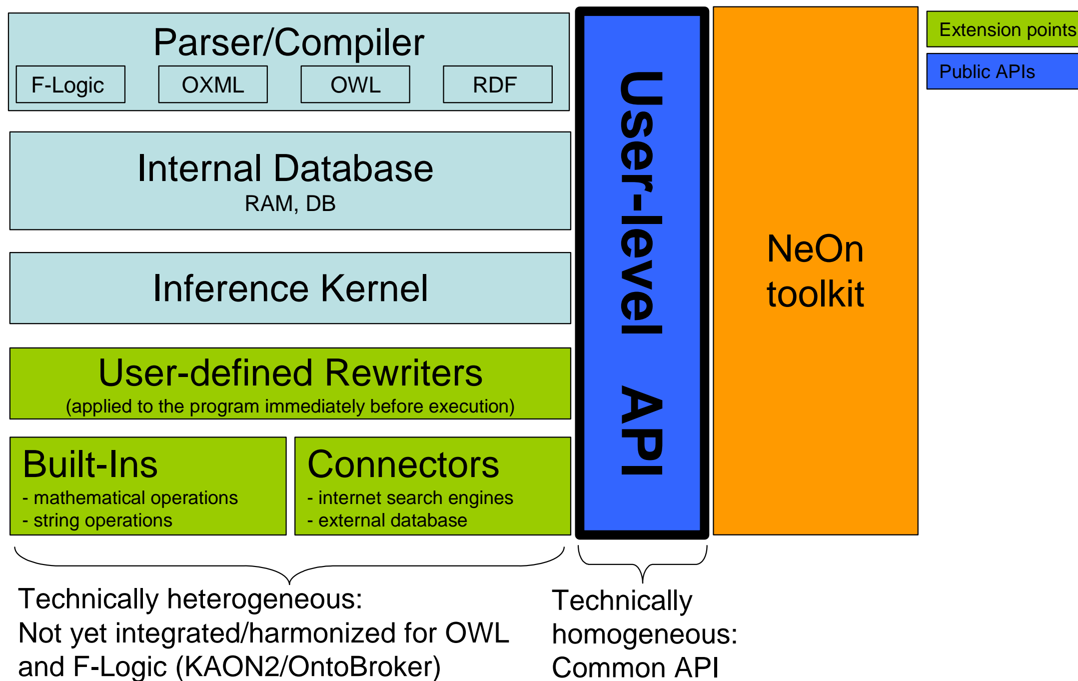

Figure 28: Current Status of Infrastructure

The figure above shows the role of the API as part of the NeOn infrastructure core. It encapsulates and bundles management and reasoning functionalities of backend components. The backend infrastructure itself is extensible. It provides the necessary degree of abstraction in order to avoid low-level method calls that are inconvenient, potentially inefficient and might lead to inconsistencies.

As mentioned above are the components implementing the API currently not harmonized with respect to their interfaces and interoperability. the figure above therefore doesn't show the existing redundancies and the differences (such as the details of the inference kernel).

5.2 API

In this section we describe the main parts of the API from a user's perspective. Most of the aspects are described on a level where there are no or only minor differences between the OWL and the F-Logic side. Later in this section we'll briefly describe some of the more specific classes.

2006–2007 © Copyright lies with the respective authors and their institutions.

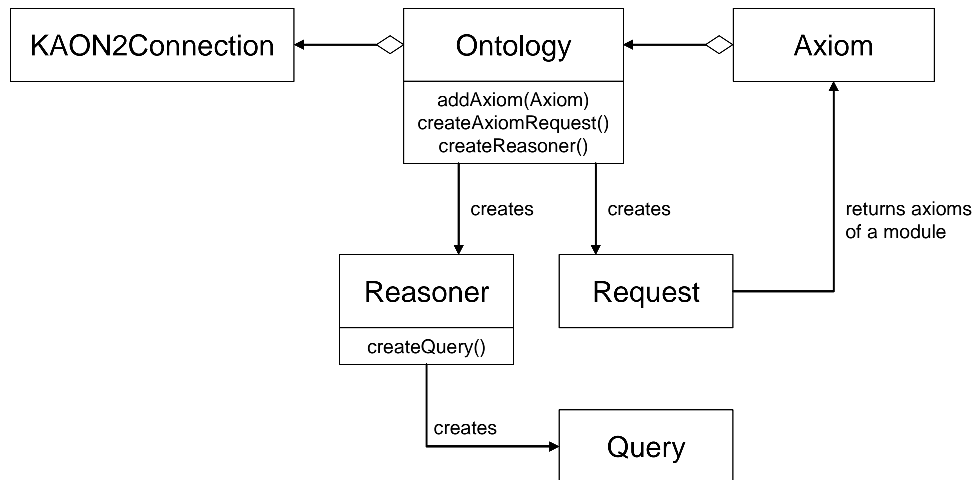

Figure 29: Ontology Management Classes

The Figure above shows the ontology management classes giving a top-level view onto the API. On there is a clear separation between the requests for asserted statements and reasoning tasks. This is important from a conceptual point of view. Reasoning tasks are potentially time- (and resource-) consuming. Designers of applications need to distinguish the two different kinds of actions, e.g. with respect to threading issues or GUI-design. Application designers can for example decide to show inferred elements only on request to clearly indicate that those elements are inferred and to avoid performance problems on the GUI-level (such as frequent mouse-clicks triggering inferencing processes).

Below is a classification of the important interfaces in the API for the ontology management level:

- NeOn common part
    - Ontology
    - Axiom
- NeOn OWL/DL model
    - Class
    - Property
- NeOn rule model
- NeOn mapping model

The NeOn model has additionally the parts mapping and modules. These are currently not directly represented in model API. Instead there are fixed language constructs in OWL and rule to represents the mappings and module functionality.

The starting-point for interactions with the system is the connection class. Via a connection object users or applications get access to ontologies.

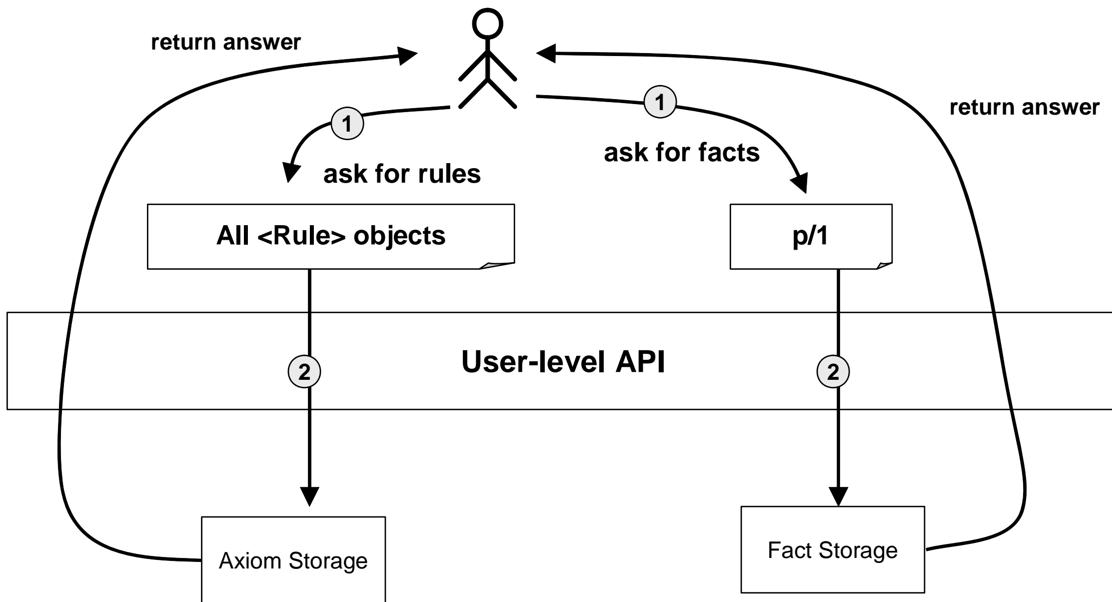

Figure 30: Handling of Requests

The Figure above shows the handling of requests via the API. Requests can be issued for schema-level data such as rules and on the instance level.

The Figure below illustrates how queries are issued through the API. Users submit queries directly on the API-level. The reasoner then controls the reasoning process, which involves rule-rewriters, the fact-store and the axiom store. All this encapsulated by the API.

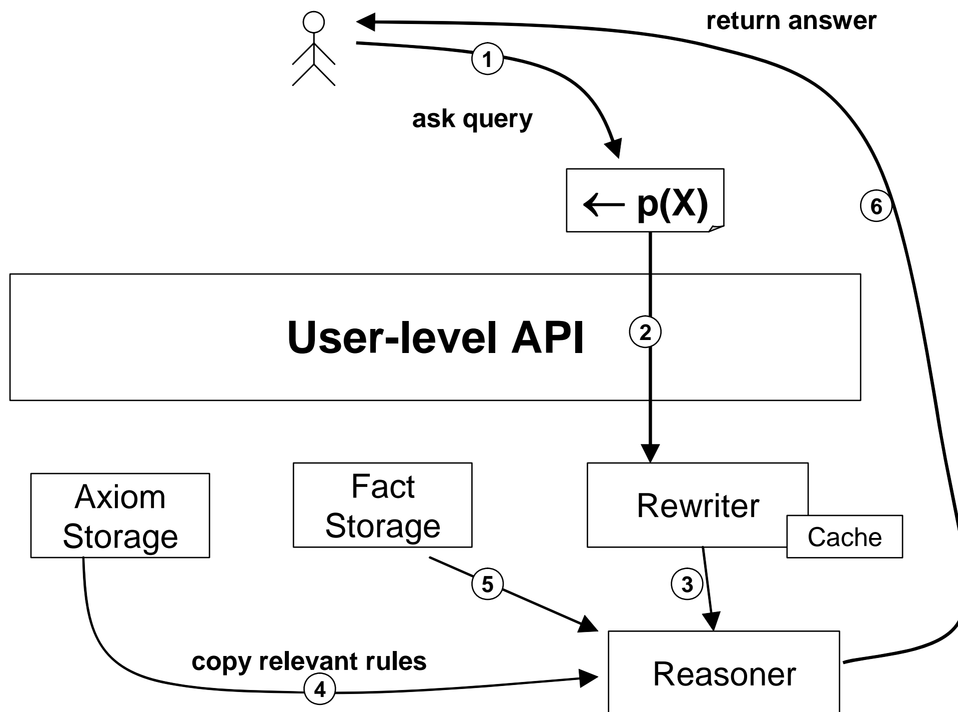

Figure 31: Handling of Queries

2006–2007 © Copyright lies with the respective authors and their institutions.

Ontology Elements

In this section we present some of the interfaces users typically interact with without describing the complete API. As mentioned above, users interact with ontologies via connections. Within a single connection an ontology is a unique object. Using request users (or applications) interact with objects that represent (OWL-)classes, axioms, properties etc.

The API layer containing the elements of the ontology language grounds on a logical layer, which is shown next two figures below.

Some of the interfaces are reflected in subsequent sections.

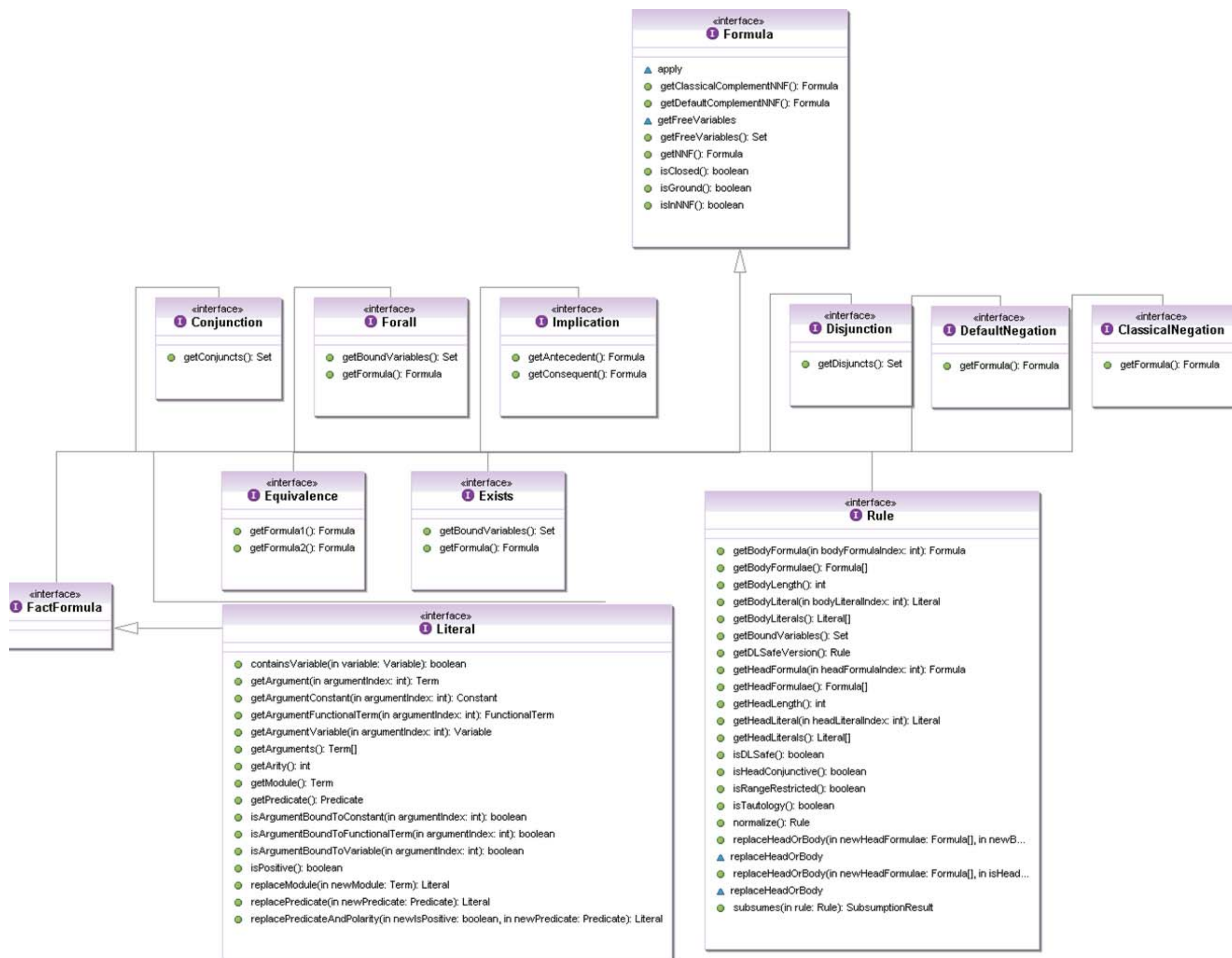

Figure 32: Interfaces of the first-order logic API (part)

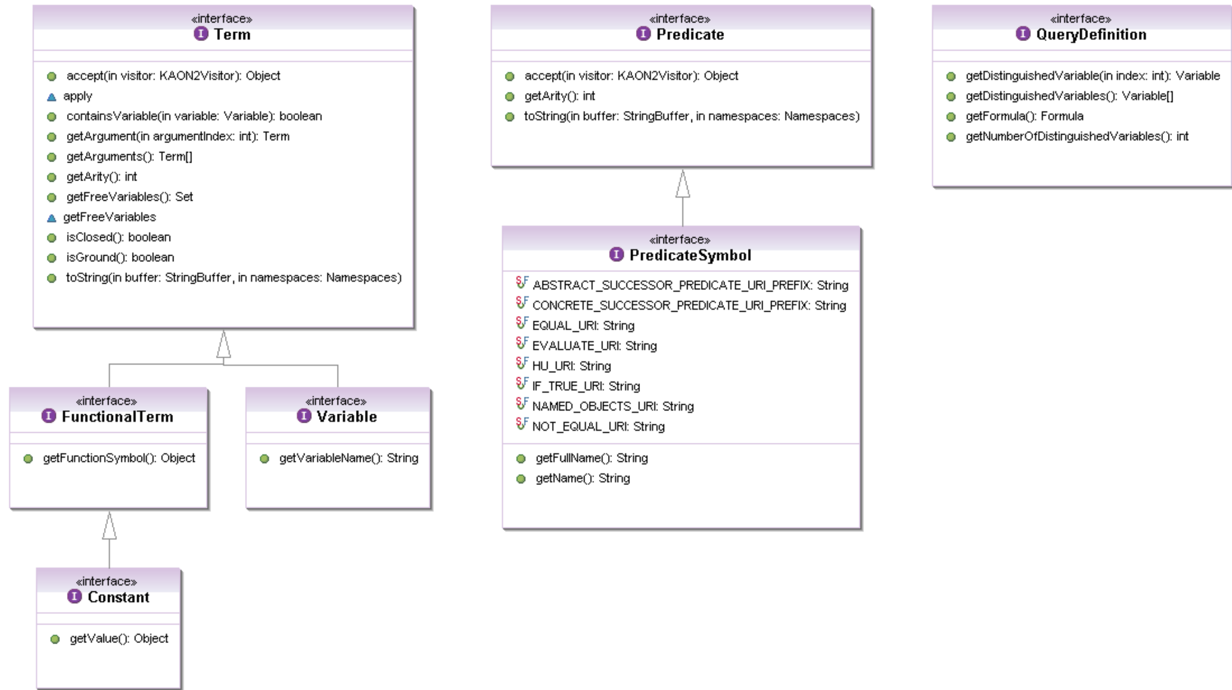

Figure 33: Interfaces of the first-order logic API (part)

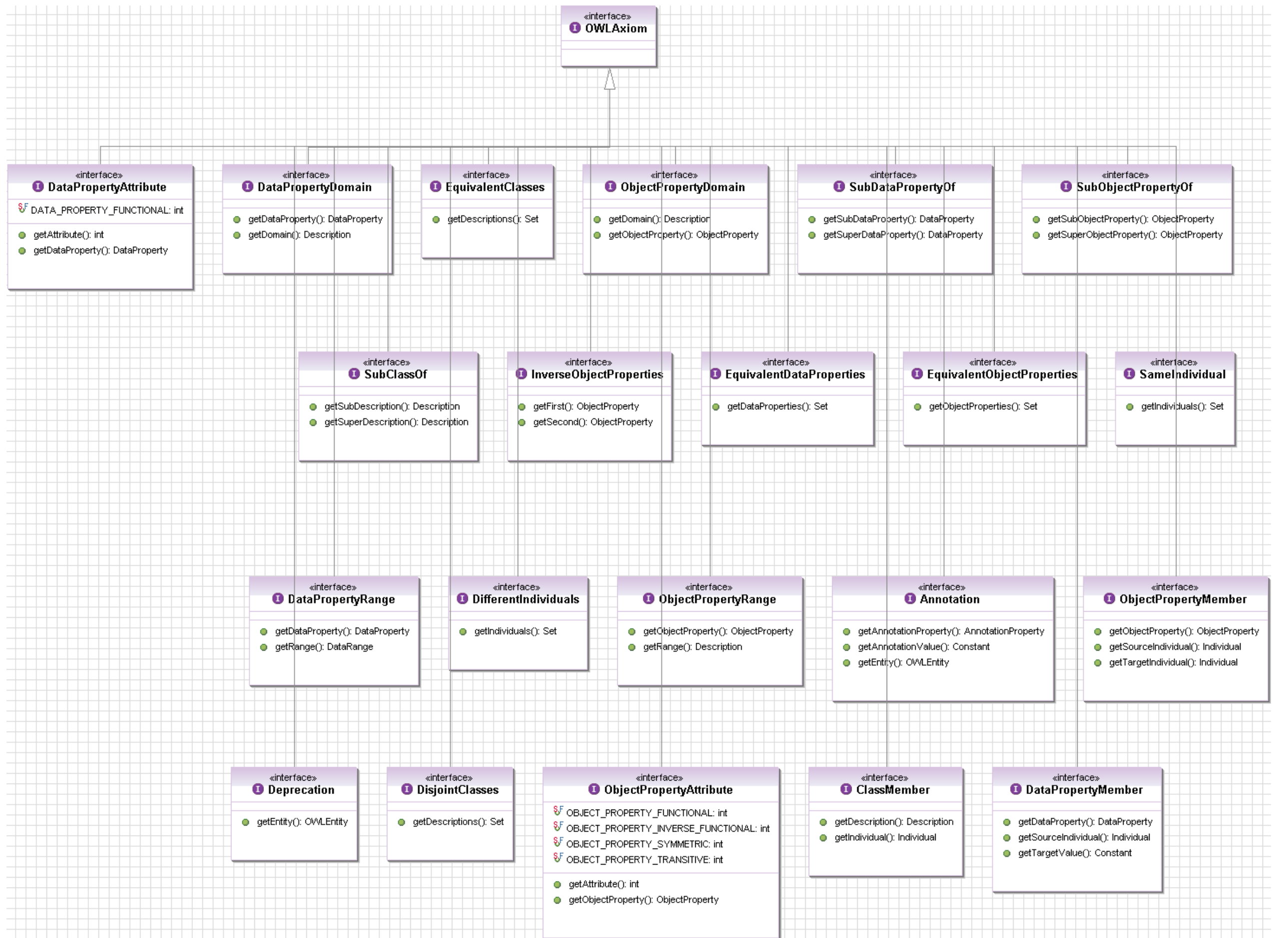

Figure 34: Axioms Interfaces

Figure 35: OWL Elements Interfaces

2006–2007 © Copyright lies with the respective authors and their institutions.

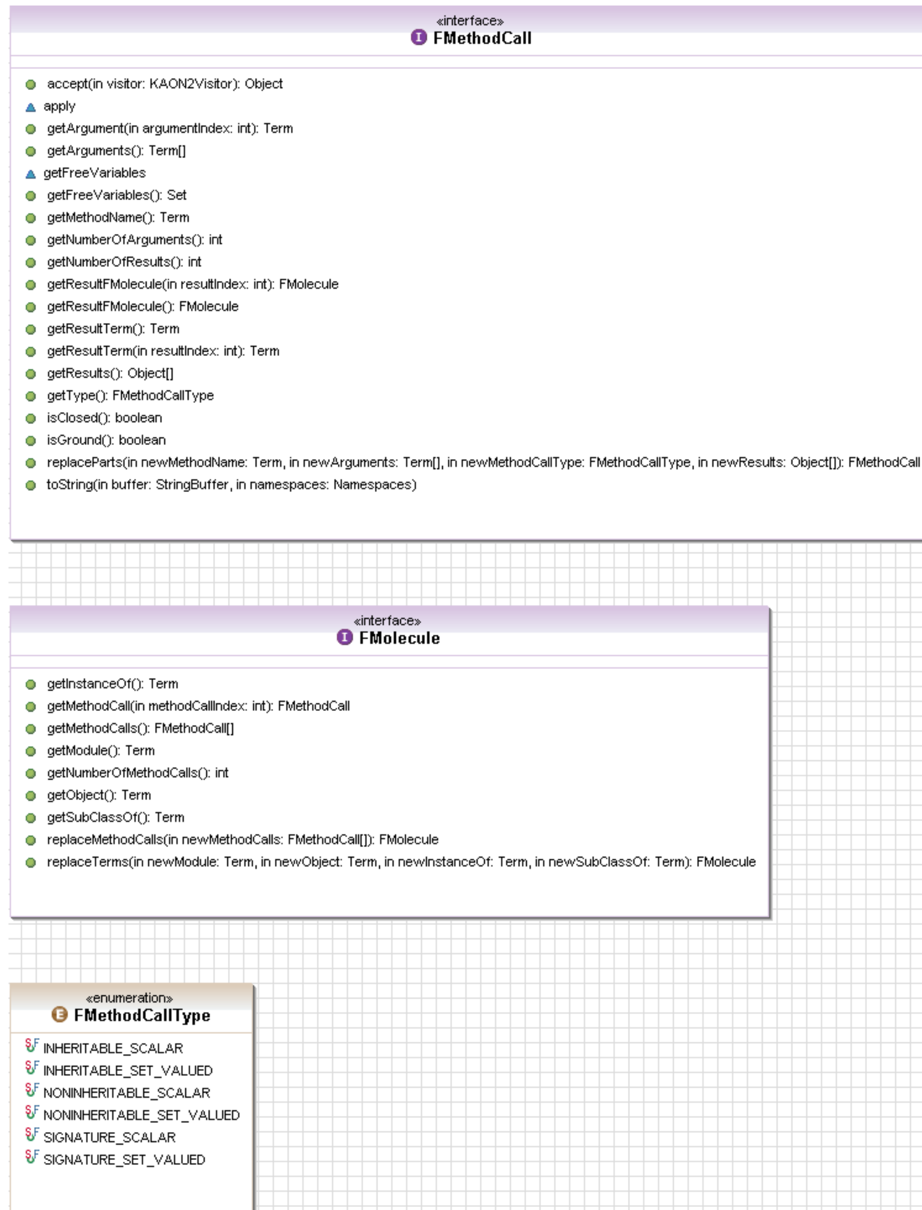

Figure 36: F-Logic Interfaces

6. Example Configurations of NeOn Architecture

6.1 Information Integrator development environment as example configuration

One of the current use cases of semantic technology is the integration of information coming from disparate, heterogeneous data sources. As figure below indicates such sources like CRM or ERP but also Sales Force Automation and Logistics have typically been developed as independent data silos or "semantic islands" within companies it now becomes more and more evident for the companies business and reporting structures to get a consolidated view on all these data.

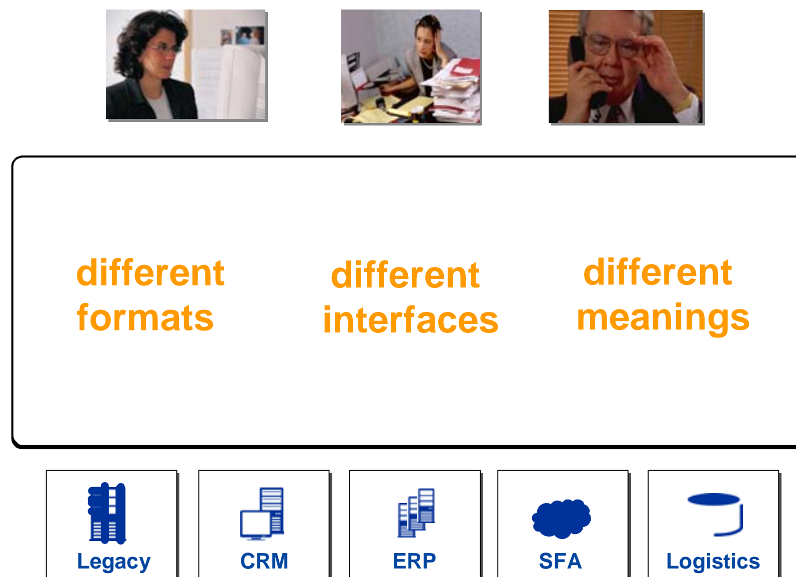

Figure 37: Application landscape

To solve the syntactic and semantic inconsistencies between these systems or to make similarities explicit semantic technology seems to be the right technology. Semantic technology connects the islands and let users understand how all the data are related to each other. However, semantic technology not only serves as a "translator" between island terminologies. In addition it can be used describe relationships and dependencies that were neither existing nor visible within single systems but which do exist across systems.

6.1.1 Ontology generation

In cases where ontologies do not already exist as such but some other means of data or information modelling already exist (UML models, database schemas, XML schemas) it might make sense to translate these models into ontologies or ontology language descriptions. These generated ontologies can serve as a starting point for further ontological engineering. They are

2006–2007 © Copyright lies with the respective authors and their institutions.

called data source ontologies. The APIs to be defined for this generation process shall allow for an easy generation of such ontologies from various types of data sources.

The figure below sketches such a generation that shall be possible for any kind of regularly structured data like SQL database but also for XML documents or Web Services.

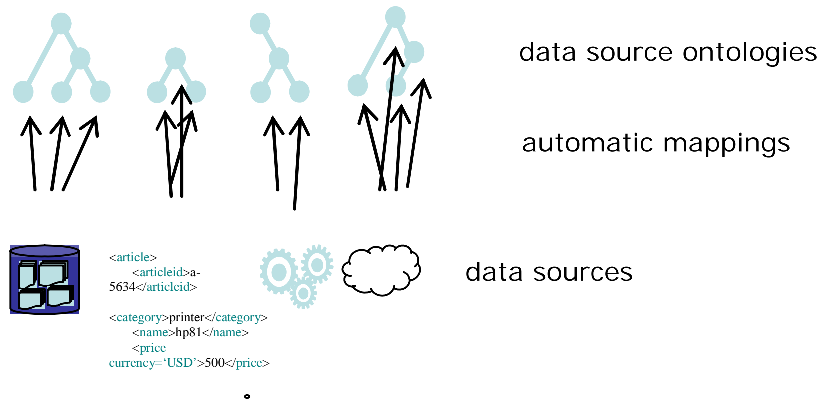

Figure 38: Generating initial ontologies

It is necessary to annotate such generated ontologies with information that is specific to the data sources used. For instance, the information about a database where a schema originates from and how to access it might be interesting for later usage of the data stored in that database. XML schema provides flexible mechanisms to allow for user-/product-specific annotations. These annotations shall somehow be considered during the transformation.

6.1.2 Mappings / Networking

In general the generated data source ontologies will not be modified as they represent the system state and nothing more. Because the goal is not to translate existing terminology into ontology terms but to connect the different systems we need additional means to express these connections. This is done via mappings which then lead to networked ontologies.

Within the Information Integrator the approach is to define higher-level integration ontologies which map to or are mapped from data source ontologies. Please note figure below shows only a single integration ontology on top of the generated source ontologies. But in principle, there might exist various levels of integration ontologies. For instance a first level might be mapped from a data source ontology and add semantic information that does not directly result for the sources but does exist therein. An example could be a mapped web service operation. While the operation itself and thus also the generated data source ontology is merely syntax, a first integration level could attach more information. Please keep in mind that it is a good policy to keep the generated data source ontology untouched. Then this enriched ontology can be used for further integration steps with other ontologies.

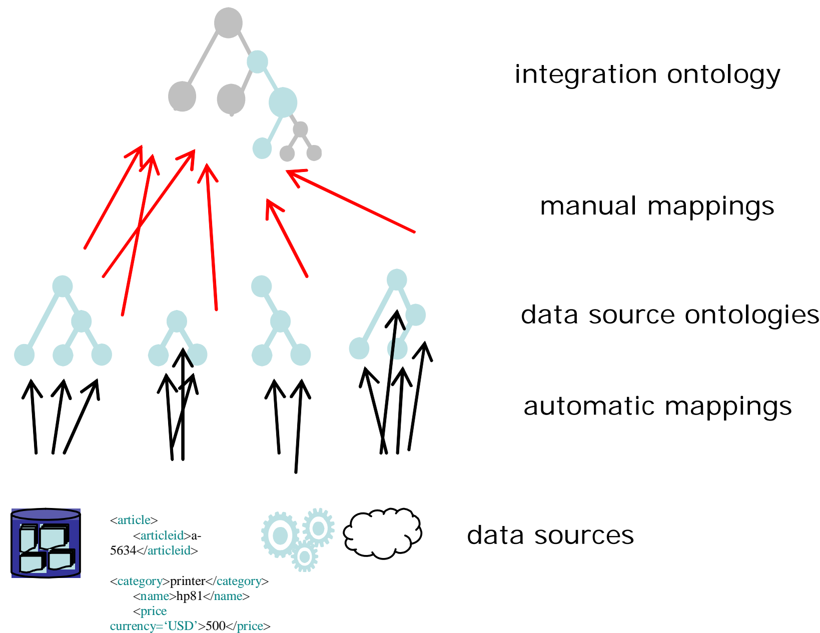

Figure 39: Integrating ontologies

The networking of ontologies can result in a network of ontologies. There might be different ways to navigate through such networks.

One possibility is to have a "normal" hierarchical ontology browser which first displays projects, then the projects' ontologies and finally the ontologies' content. A separate view can be used to look into an ontology and if the ontology references some other ontology or ontology item a double click might forward you to the referenced item.

On the other hand, like a single ontology can have many references inside and you can visualize this in a graph. You can also adopt this graph presentation to the ontology level showing ontologies as nodes and references between these ontologies as links. Thus, instead of using the hierarchical browser pattern you might feel more comfortable with a graph based presentation.

There are special use-cases where the same pattern for relationships between ontologies appears again and again. This might be enforced by some application architectures. For example as shown in figure above in the field of integrating disparate data sources you always have one layer to represent the data sources themselves. On top of this layer you may have separate models which describe the integrated model. And finally on top of the integrated model you may define views that can be used more easily by applications. From an engineers point of view a graphical representation of ontologies and other artefacts that reflects such a hierarchical structure is helpful.

In all cases where networked structures are to be represented it is essential, that the graphical presentation not only shows those items that have explicitly been loaded into the tools but it should also reflect those items that exist but might still be stored in some (remote) repository. This is a useful precondition for easy detection of related ontologies.

6.1.3 Query development

Querying ontologies is a basic functionality. Creating a query language based tooling where users can type in their queries and send these queries to reasoners is quite simple. However, the

2006–2007 © Copyright lies with the respective authors and their institutions.

experience with for instance database query tools has shown that it is essential to have some GUI based query generators (at least for beginners or occasional tool users who want to create some simple queries). On the other hand experience with the same tools has shown that it is almost impossible to create a GUI based tool that supports the entire query language without loosing the simplicity for the usage. Thus there is always a trade-off between functionality offered in the GUI and completeness. This trade-off implies that there might be different tools or at least different parameterisations for more or less experienced users.

There might be different tools for different query languages. In all these cases it should be possible to generate additional query tools. It should be possible to generate such tools from a query language API (in contrast to having a separate query model API that then has to be mapped to the query language API).

6.1.4 Deployment

Often data sources as mentioned above constitute business-critical systems. In industrial application landscape it is common practice (self-protection) to have such critical business systems run in a very controlled production environment. As a consequence applications need to be developed in a separate development environment. Furthermore before bringing a developed application into the real application landscape it is to be tested in a system that replicates the real production landscape. During these early stages the new application cannot access the “real” data sources but only systems that are specifically set up for test purposes.

That means there is a lifecycle for application systems. Within the lifecycle a system including the ontologies described above has to be deployed into different environments. The deployment process has to configure the application, i.e. has to define the concrete external data sources which are to be used in the environment where the application has to be deployed to. For this style of application development we need some lifecycle and deployment support.

6.1.5 Querying data sources

Once the application has been deployed into the final runtime environment, managers of the application landscape often demand for predictable (hopefully short) query response times. Typically these response times are estimations as measured during preceding test phases and they are meant to be almost stable even under changes in the data source content. Administrators get very nervous if requests for whatever reason require an unexpected amount of processing resources or require extended execution times. Monitoring tools aid administrators on their job to control these systems.

Consequences for the semantic query processing are twofold. First there is a strong demand in production environments for reliable query processing techniques with some optimisation steps before accessing the external data sources. In the field of data integration naïve processing strategies typically read too much data from the external data sources and consequently fail to meet the performance requirements, not only in terms of response times but also in terms of too extensive resource utilization. Necessary optimisation techniques have been heavily explored in the field of federated and distributed databases. Second, there need to be means to analyse and control the system.

6.2 Example Engineering components

6.2.1 Tightly coupled component Term Translator (UPM)

Short description of the LabelTranslator approach

LabelTranslation is a strategy and a platform created for supporting the multilingual extension of ontologies existing in just one natural language. This platform was developed within the European Esperonto project (IST-2001-34373), concluded in 2005, by the Language Technology Lab of the German Research Center for Artificial Intelligence (DFKI GmbH)[6], in Saarbrücken, and the Ontological Engineering Group at the Artificial Intelligence Laboratory of the Universidad Politécnica de Madrid[7] in Madrid, Spain. Part of this work has been partly continued within the eContent LIRICS project[8] (No. 22236), carried out at the IULATERM Group of the Universitat Pompeu Fabra in Barcelona, Spain, from 2004 until 2006.

Aims and scopes. LabelTranslator was developed in order to support "the supervised translation of ontology labels" (Declerck et al. 2006) and, at the same time, allowing for the semantic annotation of multilingual web documents using the resulting multilingual labels of ontologies. By "supervised translation" is meant, that this approach foresees the intervention of the domain expert or translator in case no results outcome, or for validating them. Therefore, LabelTranslator offers a semi-automatic strategy. LabelTranslator can be integrated into any ontology engineering platform to enable its users to translate their ontologies inside the application.

Languages involved. LabelTranslator is available for Spanish, English and German.

Steps, sources and techniques used for localizing. For the development of LabelTranslator already available multilingual semantic resources and basic natural language processing tools were reused for providing a semi-automatic translation of labels in ontologies. In the current version of the LabelTranslator platform three types of multilingual resources were included:

- EuroWordNet (EWN)[9], a semantic lexical resource.
- Wikipedia[10], the multilingual free encyclopaedia on the Web, based on knowledge of the word.
- BabelFish[11], an on-line translation service used as "fallback position" [Declerck2006].

The steps for the localizing approach are summarized in Table 5:

Table 5: Steps, sources and techniques used for localizing in LabelTranslator

Steps	Sources and techniques
1.	Upload of an ontology in the LabelTranslator platform
2.	Selection of the ontology labels to be translated in one of the target languages (en, es, de)
3.	The system accesses the EWN database for finding the selected term (or part of a term),

[6] http://www.dfki.de/web/

[7] http://www.oeg-upm.net

[8] http://lirics.loria.fr/

[9] See section 7.1 of the present document for a detailed description of the approach

[10] http://es.wikipedia.org/wiki/Wikipedia

[11] http://babelfish.altavista.com/

2006–2007 © Copyright lies with the respective authors and their institutions.

	and also checks in the WordNet database, only if the source language is English
4.	Result(s) (synset and gloss) are displayed, if the matching is successful. Users can then validate the suggestions, modify the translation and save it in the database.
5.	If the matching in EWN is not successful, the system checks in Wikipedia, which also uses a mechanism for relating entries in the various languages available
6.	If steps 3. and 5.do not provide any results, the system turns to BabelFish
7.	If still the translation is not satisfactory, the user can enter a translation, together with part-of-speech information and a definition

If the same translation session is repeated in the future, the system will return the translation already saved in the memory.

Developers of LabelTranslator give priority to the EWN resource because a "high quality in the translation is expected since "EWN has been built following semantic considerations and validated by language and/or domain experts" [Declerck2006].

Systems of representation of multilingual information . For the task of querying the data within EWN, and the retrieval of translations from it, a Java API was created [Gantner2004]. The EWN data is stored in distinct MySQL databases.. All of the databases have the same schema and can be accessed by the same SQL statements, which are contained in the monolingual API. The multilingual API consists of several objects of the monolingual API (one for each language i.e., currently English, German and Spanish), and a routine to get translations from and to any of the mentioned languages.

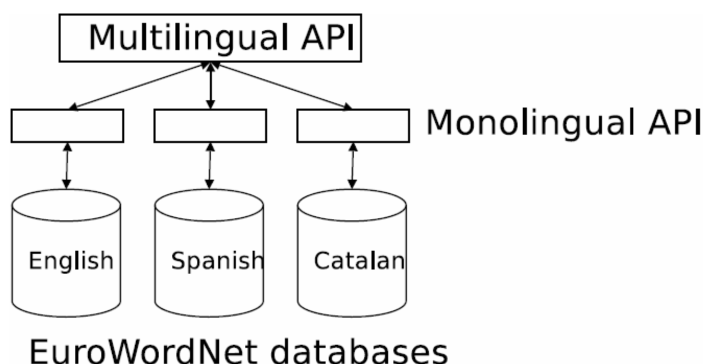

Figure 40: API structure [Gantner2004]

Architecture.

The following figure shows the general architecture of the system:

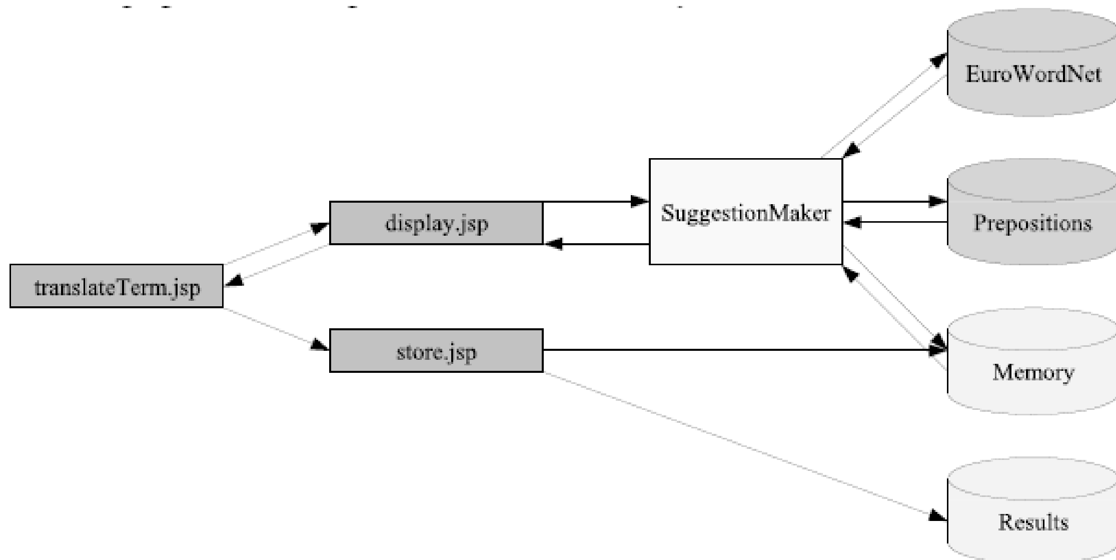

Figure 41: Architecture

The JavaServer Page **translateTerm.jsp** acts as some sort of proxy for all incoming user requests. Looking at the HTTP request parameters, it decides where to forward the request. If it is a request for a translation, it is forwarded to **display.jsp**, a page which displays some translation suggestions to the user. The user may ask for more alternatives, an action which calls again the proxy page and the display page, or select a translation that suits him, which calls the proxy page and then the page **store.jsp**. In order to assemble the suggestions, the **SuggestionMaker** component looks up possible translations in EuroWordNet (EWN), a multilingual semantic lexicon that connects identical and similar concepts in different languages. As only nouns, verbs, adjectives and some adverbs are available in EWN, there is also a component for the translation of particles, mostly prepositions. The system can also use translations that have been done before the actual translation, which are stored in the translation **memory**. The stored memory consists of both terms and EuroWordNet interlingual IDs, which represent semantic concepts (called synsets in Princeton WordNet). In EuroWordNet, if a concept (synset) in one language corresponds to a concept in another one, both have the same ID. store.jsp stores the translation result, and also puts, in some cases, the translated term, or parts of it, into the translation memory.

Loosely coupled engineering component Ontology discovery (OU)

Watson in its own right provides a scalable mechanism for discovering, selecting, and accessing ontologies that are normally distributed over the Web. While it is a standalone infrastructure, it may offer, nevertheless, several important benefits also on the level of engineering components of the NeOn reference architecture.

In particular we intend to expose two functionalities of the Watson suite: (i) the ontology discovery and selection services, and (ii) the ‘tell me more about given ontology’ service. That will be engineering components for the NeOn toolkit. In addition, we intend to collaborate and re-use some parts of the Watson to contribute to some tightly integrated components, e.g. versioning and/or modularization.

2006–2007 © Copyright lies with the respective authors and their institutions.

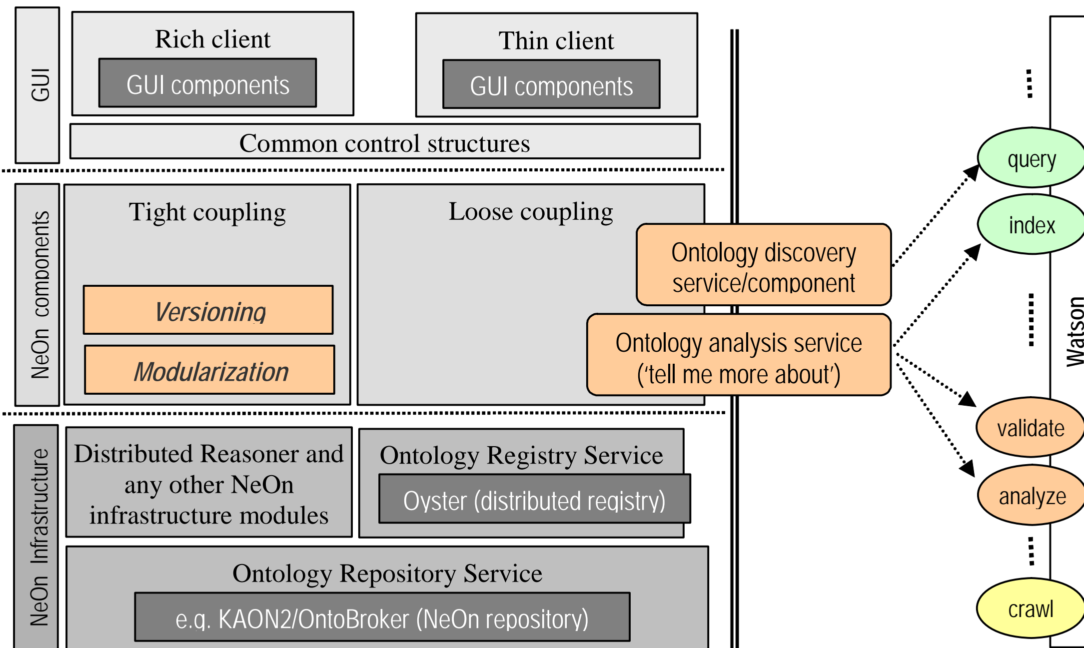

Figure 42: Watson on the level of NeOn distributed components/services

Ontology discovery / selection component

This component would enable the user of NeOn toolkit to access Watson information about discovered ontologies and use it e.g. to select ontologies containing a particular keyword, set of keyword, type of keywords, etc.

"Ontology discovery and selection" component would be a web service that opens up the keyword-based part of the querying mechanism of the Watson. We expect the signature of the service to be as follows:

```
Vector *O  getOntologiesByKeyword (
Vector String *K,
String *M,
void *C )
```

- `Vector String *K` … a vector of keywords user expects to find in the selected ontologies; may include wild cards, logical compositions, etc.
- `String *M` … a selection modifier parameter stipulating e.g. which parts of entity definition shall be included in matching the keywords, or what shall be returned as an outcome of selection (e.g. URI + keywords matched, URI + concepts matched, etc.)
- `void *C` … a function/web service enabling the user to define his or her dedicated keyword comparison function; e.g. for the purpose of matching them precisely, fuzzily, heuristically, as compounds, etc.

- `Vector *O` … a vector of ontology URIs satisfying given keywords, modifiers and comparison criteria; each URI accompanied by a vector of actual keywords/concept URIs occurring in a given ontology[12]

The functionality will be implemented as a loosely couple service. Depending on the experience with different usage scenarios and the amount of integration with other NeOn engineering components it may be later migrated to a tightly coupled component.

Ontology analysis

This component would enable the user of NeOn toolkit to access Watson information about validated and analyzed ontologies and use it e.g. to learn more about the ontologies selected by the previously described component. We expect this service being able to provide the user with information such as ontology "quality level" for a given URI, other networked ontologies related to a given URI (e.g. imports, versions, duplicates,…),

"Ontology analysis" component would be a web service that opens up the outcomes of the ontology validation and analysis mechanism of the Watson. We expect the signature of its two services to be as follows:

```
Vector *D  getOntologyAnalysisByURI (
URI *O,
Vector String *R )
```

- `URI *O` … a valid URI of an ontology stored by Watson; if URI is invalid, error state shall be returned
- `Vector String *R` … a vector of selection modifier parameters optionally stipulating which particular networked relationships or metadata shall be retrieved from the Watson DBs (e.g. 'imports', 'duplicate', 'expressivity', …)
- `Vector *D` … a vector of triples (?) associating the ontology URI from the input with additional information retrieved from analyzed DBs (dependent on the modifier)

```
Vector *D  getRelationsByConceptURI (
URI *O,
Vector URI *C,
Vector String *R )
```

- `URI *O` … a valid URI of the 'working ontology' in which the concepts are going to be processed
- `Vector URI *C` … a vector of concept URIs in a given ontology for which the user wants some additional information, definition, etc.

[12] The reason for this setup that with multiple keywords it may be only possible to satisfy query partially, hence the need to feed back the actual coverage.

2006–2007 © Copyright lies with the respective authors and their institutions.

- `Vector String *R` … a vector of selection modifier parameters optionally stipulating which particular relationships or metadata shall be retrieved from the Watson DBs (e.g. ‘parents’, ‘children’, ‘synonyms’, …)

- `Vector *D` … a vector of triples (?) associating each the concept URIs from the input with additional information retrieved from analyzed DBs (dependent on the modifier and working ontology)

We expect both of these components being only loosely coupled to the NeOn architecture – as they are mostly about accessing remote repository. As such they would need some form of GUI to present their results anyway. Hence, the actual GUI component may then be tightly integrated with the NeOn toolkit, and this GUI will use the loosely coupled component based on some remote service(s).

6.2.2 Interactive component Ontology Navigation (OU)

In addition to the loosely coupled service-based components mentioned in the previous two section, Watson team (and The Open University) is also intending to align its work with other NeOn partners. At the moment, we see two primary areas where Watson and NeOn development may benefit from alignment: (i) versioning and version management, and (ii) modularization and module management. This work is, however, beyond the existing Watson functionality, so we do not discuss it here.

As has been mentioned several times above, Watson in its own right provides a scalable and self-sufficient mechanism for discovering, selecting, and accessing ontologies via a range of querying options. While it is a standalone infrastructure, it may also offer several benefits on the level of GUI components of the rich or thin NeOn clients.

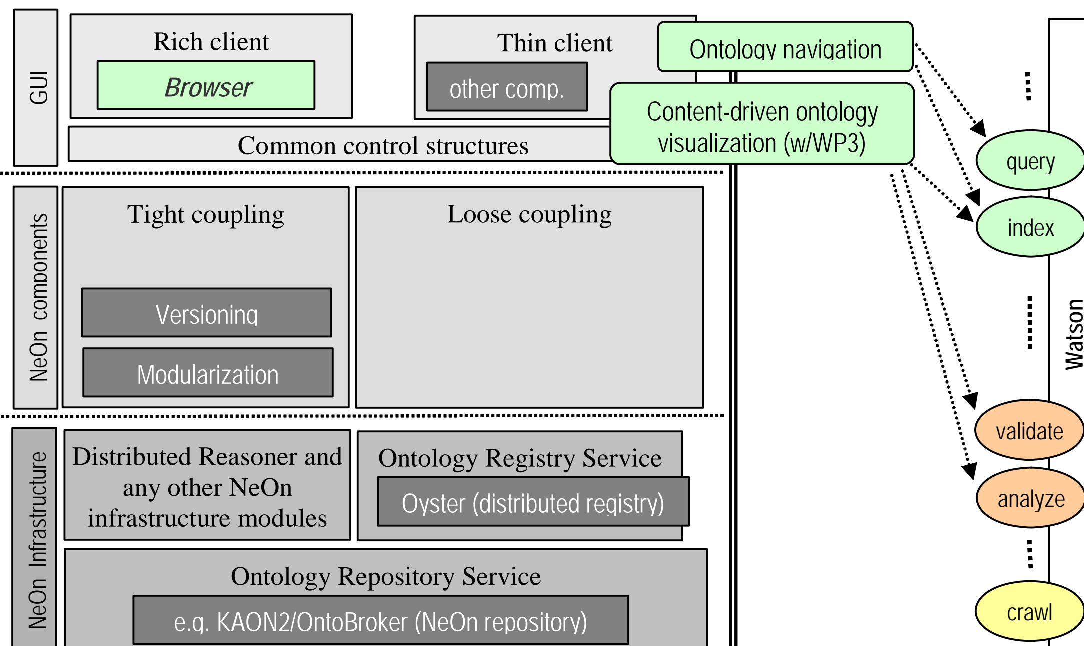

Figure 43: Watson on the level of NeOn toolkit GUIs

A similar mechanism for interacting between NeOn and Watson can be applied here as in the previous level. In particular, there would be components developed more or less as standalone

and lightweight GUI modules enabling the user (i) to navigate through the (network of) selected and retrieved ontologies, and (ii) to enable the user explore retrieved (network of) ontologies in a visual manner. Both components are, nevertheless, subject to further improvement and development within NeOn, so at this point, we do not provide more information about the intended functionality – without consulting other partners (e.g. WP3).

In addition to these standalone GUI components, there is a possibility to adapt (some of) them so that they can fit the rich NeOn toolkit. In particular, we see a component with a generic functionality of an "ontology browser" as a potential candidate for such a tighter integration. Several opportunities exist and need to be explored more in depth; potentially including:

- Contribution to the development of dedicated parts of the "ontology browser" that would provide some user interface to the loose components mentioned in the previous section (esp. specification of ontology selection conditions and presentation of analytic results)

- Adaptation of (some of) the loosely coupled GUI components to suit the rich GUI of the NeOn toolkit (and/or other NeOn architecture compatible tools); these adapted components might be used e.g. as alternative visualizations or ontology presentation techniques to complement more standard ones typically existing in the tools (e.g. hierarchies, tree views, etc.)

6.2.3 GATE as loosely coupled component (USheffield)

SAFE, the Semantic Annotation Factory Environment, addresses the NeOn context of dynamic, distributed networked annotation services. This section previews the architecture of SAFE and its integration point for NeOn. (Note that this system was previously called GLEAM, the GATE Layer for Extraction, Annotation and Mining.)

The service provides annotation functions for documents, i.e. the client of the service (other components, applications and users of the NeOn architecture) supplies text (and parameters such as ontology network, language and so on) and the service responds with semantic annotation relative to the ontology. For example a report on over-fishing and the depletion of fish stocks in coastal waters might be annotated with location, stock volume and fish species data.

The service is SOAP-based and has been developed for NeOn using Apache Axis. It is deployed server-side using Apache Tomcat and is compatible with other J2EE containers. It has been successfully trialed in an OGSA environment and will be adapted where necessary to NeOn services layer conventions.

The service hides both the parallelisation of the annotation across multiple servers and the intervention of human annotation staff (where needed) from the service client.

6.2.3.1 GATEService (aka GaS)

A GATEService is defined here as a pipeline of GATE PR's which is available via a WebService. In the SAFE architecture, it is called from an Executive. A GATEService can be a front-end to a set of GATE engines running on different servers in order to gain in scalability. The configuration of the GATEService will specify this behaviour. It is transparent to the external user of the GATEService how many machines are actually used to produce an annotation.

A GaS can declare required and optional parameters whose values are provided by the caller. The parameter values are mapped onto either feature values on the document being processed by the embedded GATE PRs, or runtime parameter values of the various PRs that make up the

2006–2007 © Copyright lies with the respective authors and their institutions.

application. The mapping is defined in a *service definition*, an XML file supplied by the GaS creator:

```
<parameters>
  <param name="depth">
    <runtimeParameter prName="MyAnnotator" prParam="maxDepth" />
  </param>
  <param name="annotatorName">
    <documentFeature name="annotator" />
  </param>
</parameters>
```

Parameter values are all strings, but when mapping to PR parameters the usual GATE `Resource.setParameterValue()` is used, which can convert strings into other types (e.g. URL, Integer, etc.).

The service definition also specifies what annotation sets the service requires as input and populates as output:

```
<annotationSets>
  <annotationSet name="Original markups" in="true" />
  <annotationSet name="" out="true" />
  <annotationSet name="Key" in="true" out="true" />
</annotationSets>
```

Note that a service can use the same set for both input and output, and can use the default annotation set (the second example above).

The traditional GATE environment will have a plugin to connect to a GATEService as to a normal GATE Application, in that case a GATEService should be able to take as input the content of a document and not only a reference to it in a DocumentService. The GaS parameters would translate to runtime parameters on the GATEServicePR.

Therefore we can distinguish 2 types of Gas :

6.2.3.1.1 GATE mode

In this mode the GaS behaves roughly like a current GATE application, i-e it takes a document and a set of parameters as input and returns a modified document. This mode is required in order to incorporate a GaS in a GATE application, like any other PR.

6.2.3.1.2 Remote DataStore

In this mode a GaS does not expect the document as input but only a **task**, i-e the location of a document on a remote DocService, taskID, location of an executive service and a set of parameter values. When the document is annotated (or processing fails), the GaS communicates directly to the executive to tell it that the task is complete (or failed). This mode is required in SAFE.

A new functionality should be created in GATE in order to be able to generate a simple service deployment (with everything running on a single server) with a couple of clicks. The result of this operation will be a WAR file, ready to be unpacked in a web application server. If you need a more advanced deployment, e.g. on a cluster, then more administrative intervention will be required.

6.2.3.2 Web-Service interface

A GATEService has the following query operations (returning string arrays):

- getRequiredParameterNames()
- getOptionalParameterNames()
- getInputAnnotationSetNames()
- getOutputAnnotationSetNames()

The main meat of the service is in the following two operations:

- processDocument (XMLcontent, parameters) returns : zero or more annotation

sets as XML

- processRemoteDocument (execLocation, taskID, dsLocation, docID, parameters, asMappings) returns : none

processDocument provides the "GATE mode" described above:

XMLcontent : a GATE document in GATE XML format. This is expected to contain at least the annotation sets required as input by the service.

parameters : an array of name/value pairs giving String values for the parameters expected by this GaS. It is an error if the parameter list does not specify a value for each required parameter name, though the value specified may be null.

It returns an array of name/value pairs where the name is an annotation set name and the value is the XML representation of that annotation set (as per `DocumentStaxUtils.writeAnnotationSet`), one entry per output annotation set name (possibly zero if, for example, the service is training a classifier and doesn't need to return any output). It is expected that the caller would replace the content of each annotation set on the original document with the content returned from the service.

processRemoteDocument provides the "SAFE mode":

execLocation : a URL (xsd:anyURI) giving the location of the executive which should be informed when the process completes (successfully or unsuccessfully)

taskID : task identifier that identifies this task to an Executive.

dsLocation : a URL giving the location of the document service in which the document to be processed resides.

docID : the ID of the document in the doc service.

parameters : as above

asMappings : an array of name/value pairs mapping the annotation sets expected by the GaS to annotation sets on the document in the doc service. A mapping is required for each input and output annotation set name the service uses. It is permitted for the same doc-service annotation set name to map to more than one GaS asName, but there is no guarantee as to the order in which output sets are written back to the doc service, so don't map two output sets onto the same

2006–2007 © Copyright lies with the respective authors and their institutions.

set in the doc service. The purpose of this mapping is to allow the same service to operate over different annotation sets, for example the results of different human annotators.

processRemoteDocument does not return anything directly - the success or failure of the operation is communicated out-of-band to the executive.

6.2.4 NeOn OWL Editor Prototype as example for NeOn Ontology meta model based generation (UKarl)

The OWL Editor OntoModel is an Eclipse-based tool for model-driven development of rule-extended ontologies and ontology mappings. It implements the MOF-based NeOn metamodel for networked ontologies and provides appropriate UML-syntax for OWL, SWRL and OWL ontology mappings (download at http://owlodm.ontoware.org/).

In order to facilitate ontology development, the following functionalities are provided:

- Ontology, rules and ontology mapping engineering utilizing a UML-like syntax, called UML-profile;
- import and export of ontologies in OWL, rules in SWRL, and OWL ontology mappings as DL-safe mappings.

The prototype that is currently being implemented is based on EMF and GMF. The OWL editor is already realized as an Eclipse plugin. The integration into the NeOn toolkit will be performed by adapting the plugin interfaces to those of the NeOn toolkit architecture.

6.2.5 FOAM (UKarl)

FOAM is a comonent for automated mapping discovery, aka ontology matching. The FOAM system is based on a generic process for ontology matching as first described in [Ehrig2004]. FOAM takes as input the reference to two ontologies, optionally a set of parameters to configure the alignment process and returns as output a set of mappings. In the current implementation, the mappings are limited to equivalences between simple ontology elements. FOAM provides three interface modes. It may be accessed through the command, via a Java API, and via an HTTP Web Service. The integration into the NeOn tollkit will be realized as a tightly coupled engineering component. The definition of the interface of the component will be aligned with the Alignment API as currently used in the alignment server developed by INRIA.

6.2.6 ORAKEL (UKarl)

ORAKEL is natural language interface (NLI) accepting as input questions formulated in natural language and returning answers from a given knowledge base. The input to ORAKEL are factoid questions starting with so called wh-pronouns such as who, what, where, which etc., but also the expressions ‘How many’ for counting and ‘How’ followed by an adjective to ask for specific values of an attribute as in “How long is the Rhein?”. Factoid in this context means that ORAKEL only provides ground facts as typically found in knowledge or data bases as answers, but no answers to why- or how-questions asking for explanations, the manner in which something happens or causes for some event.

In the ORAKEL system, we assume two underlying roles that users can play. On the one hand, we have end users of the system which interact with the system in query mode. On the other hand, domain experts or knowledge engineers which are familiar with the underlying knowledge base

play the role of lexicon engineers which interact with the system in lexicon acquisition mode, creating domain-specific lexicons to adapt the system to a specific domain.

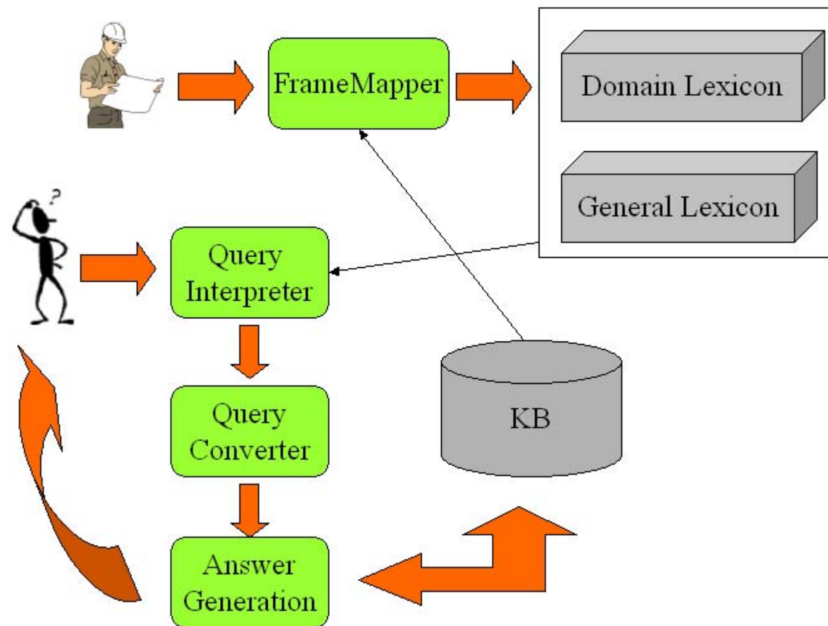

Figure 44: Overview of the ORAKEL system

The end users ask questions which are semantically interpreted by the Query Interpreter (compare figure above). The Query Interpreter takes the question of the user, parses it and constructs a query in logical form (LF), formulated with respect to domain-specific predicates. This logical form is essentially a first order logic (FOL) representation enriched with query, count and arithmetic operators. The query in logical form is then translated by the Query Converter component into the target knowledge representation language of the knowledge base, in particular to its corresponding query language. The overall approach is thus independent from the specific target knowledge language and can accommodate any reasonably expressive knowledge representation language with a corresponding query language. So far, the system has been tested with F-Logic with its query language as implemented by the Ontobroker system and OWL with the query language SPARQL as implemented by the KAON2 inference engine.

The conversion from the logical form to the target knowledge language is described declaratively by a Prolog program. The Query Converter component reads in this description and performs the appropriate transformation to the target query language. So far, we have provided the two implementations for FLogic as well as OWL/SPARQL. However, the system architecture would indeed allow to port the system to any query language.

The answer generation component then evaluates the query with respect to the knowledge base and presents the answer to the user. Answering the query is thus a deduction process, i.e. the answer to a user's question are the bindings of the variables in the resulting query. Currently, the answer generation component only presents the extension of the query as returned by the inference engine. However, more sophisticated techniques for presenting the answer to the user by describing the answer intensionally or presenting the results graphically are possible.

A crucial question for natural language interfaces is how they can be adapted to a specific domain in order to interpret the user's question with respect to domain-specific predicates.

2006–2007 © Copyright lies with the respective authors and their institutions.

In the model underlying ORAKEL, the lexicon engineer is in charge of creating a domain-specific lexicon thereby adapting ORAKEL to the domain in question. The lexicon engineer is essentially responsible for specifying how certain natural language expressions map to predicates in the knowledge base. For this purpose, we have designed an interface FrameMapper with access to the knowledge base, which supports the user in specifying by graphical means the mapping from language to relational predicates defined in the knowledge base. The result of the interaction of the knowledge engineer is a domain lexicon specific for the application in question.

For the integration of ORAKEL into the NeOn toolkit, we will develop two tightly coupled plugins that will support the two respective modes of the ORAKEL system: The query mode and the lexicon acquisition mode.

Here, the component for the lexicon acquisition (corresponding to the current FrameMapper system) will be mainly a GUI component. The component for the query mode – performing the translation of the NL query into a structured logical query and the actual query answering – will be a tightly coupled engineering component that may possibly be accompanied by a visualization plugin.

7. Mapping to Requirements

The NeOn architecture will define the organisation of components to realize the NeOn toolkit. It will contain a set of predefined components and a mechanism for the extension of ontology engineering components.

The first version of the NeOn toolkit will contain most of the predefined components. Future versions will contain more engineering components developed within the technical work packages.

Thus the requirements can be categorized in those directly realized by the NeOn architecture with the predefined components and those which are realized by user defined engineering components from other NeOn work packages.

This section describes the desirable features of the system. The philosophy of (Leffingwell & Widrig) based on the pyramid NEEDS-FEATURES-REQUIREMENTS has been used. The needs of the user are satisfied by a set of product features that are composed by a set of software requirements. The requirements are contained in the deliverable D611 [Gomez2006] of the NeOn infrastructure work package. They are here identified by their numbers in the deliverable D611.

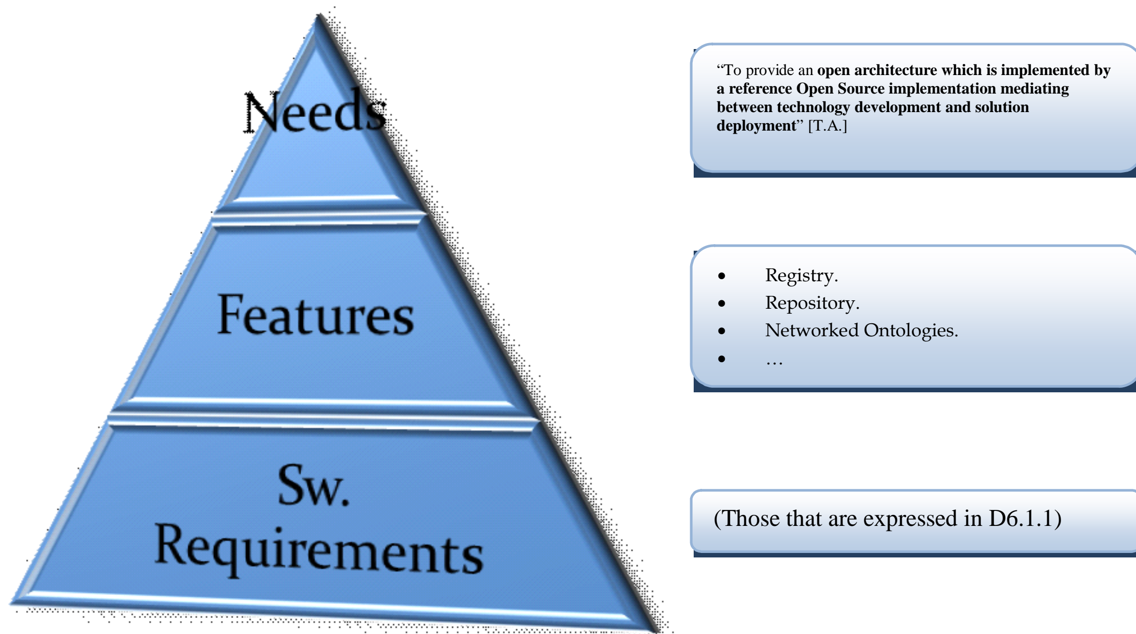

Figure 45: Pyramid [Leffingwell2003]

7.1 Features

7.1.1 Networked Ontologies

Neon takes into account the network aspect of ontologies. The architecture must deal with networked ontologies (establish mappings and relations between different ontologies). The mapping between ontologies can be done in an automated way as well as manually.

Requirements: 2.1.1.1, 2.1.2.2.1, 2.1.2.2.5, 2.1.2.2.7, 3.2.9, 3.2.9.1, 3.2.9.2, 3.2.9.3, 3.2.10, 3.2.10.1, 3.2.10.2, 3.2.11, 3.2.11.1, 3.2.11.1.1, 3.2.11.1.4, 3.2.11.1.5, 3.2.11.6

7.1.2 Registry

The network aspect of ontologies will be addressed by the integration of registry-technology, a well-established approach for networked resources. The registry will be integrated with the repository and will be a general purpose registry (not only for ontologies). Several registries will act as a federated registry. The functionality of the registry will be based on the OMV ontology meta-model. For the basic infrastructure an API is needed to query, create and manipulate ontology meta-information. This API is defined as an ebXML registry API in the form of a Java API and a web service interface.
Requirements: 2.1.1.1, 3.2.1, 3.2.1.2, 3.2.1.3, 3.2.11.1.3, 3.4.1.1

7.1.3 Repository

The NeOn repository will support large scale ontologies. The functionalities that it will offer will be: data storage, basic query mechanism, direct accesses to the ontologies and navigation through them. The data store can be a RDBMS, an XML database or the file system. The repository functionality must include session and transaction management and must deal with the multi-user aspects of NeOn.
Requirements: 2.1.1.1, 3.1.2.6, 3.1.3, 3.1.3.2, 3.1.3.2.1, 3.1.3.2.2, 3.2.1.4, 3.2.1.4.1, 3.4.1, 3.4.1.2.1, 3.4.1.2.1.1

7.1.4 Language-Model

NeOn has the goal to be compliant with common semantic web paradigms and standards. It will support F-Logic, OXML, OWL(DL), RDF, RDF(S) and CLIPS.
Requirements: 2.1.1.16, 3.5.1.5, 3.5.1.5.1, 3.5.1.5.2

7.1.5 Inference Machine

Neon will have deduction procedures to derive implicit knowledge.

- The architecture will have reasoning capabilities (automatic classification, consistency checking, etc.). The queries on the ontologies will be evaluated also on derived knowledge.
- SPARQL will be used as the query language. It requires the realization of a query processor or a mapping to a query processor like XQuery in the case of an XML ontology serialization.
- It will have a reporting tool supporting the Business Intelligence Reporting Tool (BIRT).
- OntoBroker and KAON2 will be used.
- The queries can be done using natural language using ORAKEL as the interface.

- NeOn will have rule-support in an integrated way, including debugging and profiling capabilities. The rule mechanisms must cover either F-Logic and OWL(DL) paradigms. The rule language will be SWRL. NeOn will make use of the Rule Interchange Format (RIF) standard.
  Requirements: 2.1.1.10, 2.1.1.11, 2.1.1.12, 2.1.2.3.3.3, 2.1.2.3.3.3.1, 2.1.2.3.3.3.2, 2.1.2.3.3.3.3, 3.1.2.2, 3.1.2.4, 3.2.11.1.1.1, 3.2.12, 3.2.12.1, 3.2.12.2, 3.2.13, 3.2.14, 3.2.14.1, 3.5.1.4, 3.5.1.4.1

7.1.6 Runtime support

Applications based on semantic technologies need runtime support. The infrastructure will be accessible by web services. It will be dynamically and remotely managed. Applications developed with the NeOn toolkit can be deployed outside its Graphical User Interface (GUI) using programmatic access via the NeOn API and web services.
Requirements: 2.1.1.2, 2.1.1.4, 2.1.1.15, 2.1.1.5, 2.1.1.1, 3.2.5, 3.5.1.1, 3.5.1.2, 3.5.1.3, 3.5.1.3.1, 3.5.1.3.3

7.1.7 Integrated Development Environment (IDE)

The NeOn toolkit includes editors such textual and graphical editors. It also includes viewers or GUI components for displaying non-editable information such the metadata of a file; as well as debugging tools (for example for rules) and error handling capabilities. It also offers a concept to organize the visualization and the manipulation of the data model. The textual editors must include features like auto-completion and syntax and semantics checking. The viewers must visualize concept taxonomy, instance view, graphical ontology viewer, query tabs, etc. It will make use of the Ontology Navigator and the Entity Properties View.
Requirements: 2.1.1.3, 2.1.2.1, 2.1.2.2, 2.1.2.2.8, 2.1.2.3.3, 2.1.2.3.5, 3.2.11.1.2

7.1.8 Ontology Discovery and Analysis

To provide a mechanism for discovering, selecting, and accessing ontologies that are normally distributed over the Web. Also to provide the user with information such as ontology "quality level" for a given URI, other networked ontologies related to a give URI (e.g. imports, versions, duplicates ...). The ontology registry will provide that functionality.

Requirements: 2.1.1.8, 2.1.2.2.9, 2.1.2.2.9.1, 2.1.2.2.9.3, 2.1.2.3.3.4, 3.2.2.4, 3.2.2.5, 3.2.7

7.1.9 Usability

Ease of use for helping the user build the ontology easily using menus and wizards.
Requirements: 2.1.2.1.1, 2.1.2.1.3, 2.1.2.2.6, 2.1.2.3, 3.2.3, 3.2.3.1

7.1.10 Context and customization

Possibility to understand an ontology and its associated resources according to a given context. A way of customization is to define of the arrangement on viewers and editors that can be adapted by the user. Another one is to have the possibility to select different perspectives on resources. The engineers can customize the environment for their specific needs.

2006–2007 © Copyright lies with the respective authors and their institutions.

Requirements: 2.1.1.9, 2.1.2.1.2, 2.1.2.1.2.1, 2.1.2.2.2, 2.1.2.2.9.2, 2.1.2.3.1, 2.1.2.3.3.5, 3.2.8, 3.2.16, 3.2.16.1, 3.2.16.2

7.1.11 Lifecycle support and Collaboration

Ontologies evolve dynamically rather than in a waterfall-like manner. They are developed, changed and reused. Ontologies are merged or modularized. The semantic applications can be developed in a collaborative way. Team support is needed (workflow, collaboration, documentation, versioning). Subversion functionality will be realized on top of WebDAV based locking facilities DeltaV.
Requirements: 2.1.1.13, 2.2.2.2.3, 2.1.2.2.4, 2.1.2.3.3.6, 2.4.1.4.2.1, 3.2.1.4.1, 3.2.1.4.1.1, 3.2.6, 3.2.11.1.7, 3.2.11.1.7.1, 3.2.11.1.7.1.1, 3.2.11.1.7.1.2, 3.2.11.1.7.1.3, 3.2.11.1.7.1.3.1, 3.2.11.1.7.2, 3.2.11.1.7.2.1

7.1.12 Heterogeneity

It will be possible to integrate heterogeneous components in large applications. The infrastructure must cover different paradigms as well as different aspects of semantic applications.
Requirements: 2.1.1.14, 2.1.1.17, 3.1.2, 3.4.1.2

7.1.13 Modularity/Extensibility

The architecture can be extended by its plug-in capabilities. The architecture of NeOn is Service Oriented (SOA).
Requirements: 2.1.1.18, 2.1.1.19, 2.1.1.19.1, 2.1.1.19.2, 2.1.2.3.2, 2.1.2.3.2.1, 2.1.5.1, 3.1.1, 3.1.1.1, 3.1.1.2, 3.1.1.3, 3.6.4.1, 3.6.4.1.1, 3.6.4.1.2

7.1.14 Integration with the "Non-Semantic World"

NeOn will be integrated with RDBMS (e.g. Oracle, MySQL, DB2), XML, etc. Dynamic integration of data sources, database schema imports; mapping and annotation tools are needed.
Requirements: 2.1.1.5, 2.1.2.2.10, 3.2.2, 3.2.2.1, 3.2.2.2, 3.2.2.2.1, 3.2.2.3, 3.2.4, 3.2.4.2, 3.4.1.2.2, 3.4.1.2.2.2, 3.4.1.2.2.2.1, 3.4.1.2.2.2.2

7.1.15 Unstructured Information Management

NeOn will have text engineering capabilities as well as an annotation service.
Requirements: 2.1.1.5, 2.1.1.6, 2.1.1.7, 2.1.2.2.10, 2.1.2.3.3.1, 2.1.2.3.3.2, 3.1.2.5, 3.2.2, 3.2.4, 3.2.4.1, 3.2.4.3, 3.4.1.2.2.1

7.1.16 Multilinguality

The architecture must deal with multingual ontologies as well as include translation services.
Requirements: 3.2.15

7.1.17 Security

The architecture will prevent malicious access to NeOn data or functionality from occurring in the NeOn framework with the aim to avoid system malfunction and unexpected behavior.
Requirements: 2.1.2.3.4, 2.1.2.3.4, 2.1.2.3.4.2, 3.6.3.1, 3.6.3.1.1, 3.6.3.1.2

7.1.18 Web Integration

Integration with common web interfaces.
Requirements: 2.1.2.3.6, 3.5.1.3.1.1

7.1.19 Multiplatform

The NeOn toolkit will run on different platforms, at least on Windows, Linux and MacOS.
Requirements: 2.1.3.1, 2.1.3.2, 2.1.4.1, 2.1.4.1.1, 2.1.4.1.1.1, 2.1.4.1.1.2, 2.1.4.1.1.3, 2.1.4.1.2

7.1.20 Open Source

The essential functionality of the NeOn toolkit on the side of ontology engineering will be available as an open source version. This does not include the inferencing core.
Requirements: 2.4.1.5.2.2, 2.4.2, 2.4.2.1

7.1.21 Documented

To make a special effort and incorporate high quality documentation and help that allows users to fully grasp all the possibilities offered by the software.
Requirements: 3.7.1, 3.7.1.1, 3.7.1.2, 3.7.1.3, 3.7.2, 3.7.2.1, 3.7.2.2

2006–2007 © Copyright lies with the respective authors and their institutions.

8. Trademarks

CentraSite®

OntoBroker®

OntoStudio®

9. References

Abney1996 S. Abney, S. (1996), Partial Parsing via Finite State Cascades Language Engineering, no. 4, vol. 2, pp.337-344

Bird2000a Bird, S, Day, D., Garofolo, J., Henderson, J., Laprun, C. & Liberman, M. ATLAS: A Flexible and Extensible Architecture for Linguistic Annotation. In Proceedings of the Second International Conference on Language Resources and Evaluation, 1699-1706, 2000.

Clayberg2006 Clayberg, E., Rubel, D., Eclipse: Building Commercial-Quality Plug-ins, 2nd Edition, Addison Wesley Professional, 2006

Clemm2002 Clemm et al., Web Distributed Authoring and Versioning (WebDAV) Versioning extensions, IETF Request for comments 3253, http://www.ietf.org/rfc/rfc3253.txt

Clemm2004 Clemm et al., Web Distributed Authoring and Versioning (WebDAV) Access Control Protocol, IETF Request for comments 3744, http://www.ietf.org/rfc/rfc3744.txt

Declerck2005 Declerck, T. & O. Vela (2005). “LabelTranslator: Multilingualism in Ontologies”. *Proceedings of the 4th International Semantic Web Conference. 2005.*

Declerck2006 Declerck, T., A. Gómez-Pérez, O. Vela, Z. Gantner, D. Manzano-Macho (2006). “Multilingual Lexical Semantic Resources for Ontology Translation”. *Proceedings of LREC 2006*.

Ehrig2004 Marc Ehrig, York Sure: Ontology Mapping - An Integrated Approach. ESWS 2004: 76-91, 2004

Gantner2004 Gantner, Z. (2004). *TermTranslation – A Tool for the Semiautomatic Translation of Ontologies*. Technical report written at the Ontology Engineering Group of the UPM, Spain [unpublished].

Ginsberg2006 Ginsberg A., Hirtle D., McCabe F., Patranjan P., RIF use cases and requirements, Working Draft W3C 2006

http://www.w3.org/TR/rif-ucr/

Goland et al 1999 Goland et al., HTTP Extensions for Distributed Authoring – WebDAV, IETF Request for comments 2518, http://www.ietf.org/rfc/rfc2518.txt

2006–2007 © Copyright lies with the respective authors and their institutions.

Gomez1999 GOMEZ, A.; MORENO, A.; PAZOS, J.; SIERRA-ALONSO, A.: Knowledge maps: An essential technique for conceptualisation. *Data & Knowledge Engineering*, Vol. 33 (2000), pp 169-190

Gomez2006 Gómez-Pérez J.M, Buil C, Muñoz, O.Requirements on NeOn architecture. NeOn Deliverable D6.1.1, 2006.

Grau2006 Grau B. C., Motik B., Patel-Schneider P., OWL 1.1 Web Ontology Language, XML syntax, W3C Note 2006

http://www.w3.org/Submission/owl11-xml_syntax/

http://www.w3.org/Submission/owl11-xml_syntax/schema/owl1.1.xsd

Grishman97 Ralph Grishman: Information Extraction: Techniques and Challenges, in International Summer School, SCIE-97, Frascati, Italy, 14-18, 1997. Lecture Notes in Computer Science 1299 Springer 1997, ISBN 3-540-63438-X

Ide2000 Ide, N. and Bonhomme, P. and Romary, L. (2000), XCES: An XML-based Standard for Linguistic Corpora
Proceedings of the Second International Language Resources and Evaluation Conference (LREC)}, Athens, Greece, pp. 825--830

IEEE2000 IEEE Standard 1471, Recommended Practice for Architectural Description of Software-Intensive Systems, 2000

LiMa2006 Li Ma, GuoTong Xie, Yang Yang, Lei Zhang, IBM Integrated Ontology Development Toolkit, IBM Alphaworks 2006, http://www.alphaworks.ibm.com/tech/semanticstk

Kifer1995 Michael Kifer, Georg Lausen, James Wu: Logical Foundations of Object-Oriented and Frame-Based Languages. Journal ACM 42(4): 741-843 (1995)

Leffingwell2003 Leffingwell, D., & Widrig, D. *Managing Software Requirements – A Use Case Approach.* Addison-Wesley, 2003.

LiMa2006 Li Ma, GuoTong Xie, Yang Yang, Lei Zhang, IBM Integrated Ontology Development Toolkit, IBM Alphaworks 2006, http://www.alphaworks.ibm.com/tech/semanticstk

McEnery2000 A.M. McEnery, A.M. and Baker, P. and Gaizauskas, R. and Cunningham, H. (2000), EMILLE: Building a Corpus of South Asian Languages
Vivek, A Quarterly in Artificial Intelligence}, no.3, vol. 13, pp. 23-32

Motik2006a	Motik B., Sattler U., A Comparison of Reasoning Techniques for Querying large Description Logic A-Boxes, in Proceedings of Logic for Programming, Artificial Intelligence, and Reasoning, Springer, 2006
Motik2006b	C., Motik B., Patel-Schneider P., Horrocks I., OWL 1.1 Web Ontology Language, Structural Specification and Functional-style Syntax, W3C Note 2006 http://www.w3.org/Submission/owl11-owl_specification/
Neff2004	Neff, M. S. and Byrd, R. J. and Boguraev, B. K. (2004),The Talent System: TEXTRACT Architecture and Data Model, Journal of Natural Language Engineering
OSGI2003	OSGi Service Platform, Release 3, IOS Press 2003, ISBN 1-58603-311-5
OSGI2005	OSGi Alliance, About the OSGi Service Platform, Technical Whitepaper 4.1, 2005
SWRL2004	Horrocks, I. et al., SWRL: A Semantic Web Rule Language - Combining OWL and RuleML, W3C Member Submission, 2004
Tablan2002	Tablan, V. and Ursu, C. and Bontcheva, K. and Cunningham, H. and Maynard, D. and Hamza, O. and McEnery, T. and Baker, P. and Leisher (2002), M., A Unicode-based Environment for Creation and Use of Language Resources 3rd Language Resources and Evaluation Conference, Las Palmas, Canary Islands, Spain
Weiten2006	Weiten; Maier-Collin, M.; Angele, M.; J.: D4.5.4 Prototype of the ontology mediation software V2. SEKT Project Deliverable 2006
Wund2004	WUNDERLICH, L.: Eclipse vs. Netbeans. *JavaMagazin* (2004) No. 5

2006–2007 © Copyright lies with the respective authors and their institutions.