



D4.2 Annex: **Heterogeneous Knowledge Store Software Companion**

Zlatina Marinova (Ontotext Lab, Sirma Group Corp.),
Atanas Ilchev (Ontotext Lab, Sirma Group Corp.),
Vasil Momtchev (Ontotext Lab, Sirma Group Corp.),
Stefan Vatev (Ontotext Lab, Sirma Group Corp.)

Abstract

EU-IST Specific targeted research project (STREP) IST-2004-026460 TAO
Deliverable D4.2 (WP 4)

In this deliverable we present the Heterogeneous Knowledge Store (HKS), a semantic repository supporting heterogeneous knowledge management, while providing scalability and efficiency to match enterprise needs. First we discuss the requirements that have driven the HKS functionality definition, then the architecture and implementation decisions. HKS currently provides functionality for storage, retrieval and querying data coming from: textual documents, results of content augmentation, ontologies, semantic web services.

Keyword list: heterogeneous knowledge, storage, legacy content, query, SPARQL, ontology, semantic annotations, SAWSDL

WP4 **Scalable, Heterogeneous Knowledge Stores** Document ID: <TAO/2008/D4.2/v1.0>

Nature: Report

Dissemination: PU

Contractual date of delivery: new deliverable, Actual date of delivery: **May 9, 2008**
accompanying D4.2

Web links: <http://www.tao-project.eu/resources/publicdeliverables/ORDI-distrib.zip>,
<http://ordi.sourceforge.net/>

TAO Consortium

This document is part of a research project partially funded by the IST Programme of the Commission of the European Communities as project number IST-2004-026460.

University of Sheffield

Department of Computer Science
Regent Court, 211 Portobello St.
Sheffield S1 4DP
UK
Tel: +44 114 222 1930
Fax: +44 114 222 1810
Contact person: Kalina Bontcheva
E-mail: K.Bontcheva@dcs.shef.ac.uk

Mondeca

3, cité Nollez
75018 Paris
France
Tel: +33 (0) 1 44 92 35 03
Fax: +33 (0) 1 44 92 02 59
Contact person: Jean Delahousse
E-mail: jean.delahousse@mondeca.com

University of Southampton

Southampton SO17 1BJ
UK
Tel: +44 23 8059 8343
Fax: +44 23 8059 2865
Contact person: Terry Payne
E-mail: trp@ecs.soton.ac.uk

Sirma Group Corp., Ontotext Lab

Office Express IT Centre, 5th Floor
135 Tsarigradsko Shosse Blvd.
Sofia 1784
Bulgaria
Tel: +359 2 9768 303
Fax: +359 2 9768 311
Contact person: Atanas Kiryakov
E-mail: naso@sirma.bg

Atos Origin Sociedad Anonima Espanola

Dept Research and Innovation
Atos Origin Spain, C/Albarracin, 25, 28037
Madrid
Spain
Tel: +34 91 214 8835
Fax: +34 91 754 3252
Contact person: Jaime García Sáez
E-mail: jaime.2.garcia@atosorigin.com

Dassault Aviation SA

DGT/DPR
78, quai Marcel Dassault
92552 Saint-Cloud
Cedex 300
France
Tel: +33 1 47 11 53 00
Fax: +33 1 47 11 53 65
Contact person: Farid Cerbah
E-mail: Farid.Cerbah@dassault-aviation.com

Jozef Stefan Institute

Department of Knowledge Technologies
Jamova 39
1000 Ljubljana
Slovenia
Tel: +386 1 477 3778
Fax: +386 1 477 3131
Contact person: Marko Grobelnik
E-mail: Marko.Grobelnik@ijs.si

List of Abbreviations

FTS	Full Text Search
GATE	General Architecture for Text Engineering
HKS	Heterogeneous Knowledge Store
KIM	Knowledge and Information Management platform
ORDI	Ontology Representation and Data Integration Framework
OWL	Web Ontology Language
RDF	Resource Description Framework
SAWSDL	Semantic Annotations for WSDL
SPARQL	SPARQL Protocol and RDF Query Language
SWS	Semantic Web Service
TRREE	Triple Reasoning and Rule Entailment Engine
URI/L	Uniform Resource Identifier/Locator
WSDL	Web Service Definition Language
WSMO	Web Service Modeling Ontology
WSML	Web Service Modeling Language

Executive Summary

One of the major prerequisites for the realization of the TAO transitioning methodology and infrastructure is the implementation of a common storage for all kinds of knowledge involved in the transitioning process. Such a repository shall offer the possibility to uniformly store content, semantic annotations, annotated Web services, ontologies and non-textual documents. It shall also provide support for unified query over stored knowledge in order to support semi-automatic transitioning. Given the fact that big volumes of data can be associated with a legacy application, the heterogeneous knowledge store shall feature highly scalable and efficient implementation.

A store that is capable of persisting knowledge in a unified format, as well as of querying across knowledge types can also be used for solving other enterprise problems such as: governance, unified and meaningful search over diverse content, knowledge management, etc.

In order to provide such functionality, a repository shall enable representation of different types of knowledge into a common, unified format; shall provide efficient storage, loading and querying of data; and shall be very scalable with respect to the volume of information being stored. Traditional repositories exist, that have proven to work efficiently and provide good scalability to accommodate huge volumes of data. Nevertheless, they lack dynamics needed when representing knowledge whose structure may change in time. Such knowledge for example is represented in ontologies, which are difficult to maintain in traditional databases (see [1]). These needs have led to the development of a new class of storage – semantic repositories.

Semantic repositories on the other hand are typically tightly connected to a particular knowledge representation format (RDF¹, in most cases) and are incapable of storing data represented in different formats. Other limitations are the scalability and efficiency of storage, retrieval and querying of semantic data.

In this deliverable we present the Heterogeneous Knowledge Store (HKS), a semantic repository supporting heterogeneous knowledge management, while providing scalability and efficiency to match enterprise needs. First we discuss the requirements that have driven the HKS functionality definition, then the architecture and implementation decisions.

The implemented Heterogeneous Knowledge Store currently provides functionality for storage, retrieval and querying data coming from: textual documents, results of content augmentation, ontologies, semantic web services.

Some of the components used for the implementation of the Heterogeneous Knowledge Store have been jointly developed with other projects and are hence partially funded by TAO. More specifically:

- TRREE and ORDI model implementation are a joint development of TAO and TripCom². TRREE has been reimplemented to provide native support for the rich ORDI data model (previously it supported RDF). This was supported by both projects, while specifically for the purposes of TAO, a FTS functionality is being developed in TRREE.

¹ <http://www.w3.org/RDF/>

² <http://www.tripcom.org/>

D4.2-Annex/ Heterogeneous Knowledge Store Software Companion

- Data Services – ORDI data services used in HKS are developed in TAO, except for the wsmo4rdf data service which is jointly developed in the SemanticGov and TAO projects.
- HKS client interface – fully developed in TAO

Due to the loosely-coupled, modular architecture of HKS it provides us with different options for exploitation. First, the HKS will be exploited as part of the TAO Suite to serve as storage throughout the transitioning of legacy applications.

Additionally, data services created for HKS can be used separately or in combination to solve problems not falling within the TAO scope. One such example is the SAR data service which will be exploited as storage for purposes of knowledge management. It can also be used in combination with the Ontology Repository data service for ontology-based content augmentation tasks. Another such possibility is to combine the Ontology repository and SWS Repository data services for applications dealing with modelling and management of semantic web services.

Our plans for HKS development in the third year of the TAO project include:

- implement native storage for SAWSDL (translation from SAWSDL to RDF);
- improvements of full-text search support;
- evaluation of HKS and its sub-components, which will be reported in deliverable D4.4;
- authentication functionality;
- integration with the distributed indexing and query component ;
- changes and customizations needed for the purposes of integration and usage by the two case studies.

Other enhancements of HKS outside of the TAO scope include:

- HKS configurability – to support usage of different data services in different scenarios;
- extensions for other types of knowledge – implement other data services;
- UI for administration and management of repository data.

Contents

TAO Consortium	2
List of Abbreviations	3
Executive Summary	4
Contents	6
1 Introduction.....	8
1.1 Relevance to TAO.....	8
1.1.1 Relevance to project objectives	8
1.1.2 Relation to other workpackages.....	9
1.2 Deliverable Outline.....	9
2 Heterogeneous Storage Requirements	10
2.1 Transitioning Process Requirements.....	10
2.1.1 Legacy Data Uploading	10
2.1.2 Ontology Generation.....	10
2.1.3 Content Augmentation.....	11
2.1.4 Semantic Annotation of Web Services	12
2.2 Requirements for General Usage.....	12
2.3 Conclusions.....	13
3 HKS Design.....	14
3.1 Functional Architecture	14
3.1.1 Document Repository	15
3.1.2 Ontology Repository.....	15
3.1.3 Semantic Annotations Repository.....	16
3.1.4 Semantic Web Service Repository.....	17
3.1.5 Query Support.....	17
3.2 HKS interfaces	17
3.2.1 Management of Legacy Documents	18
3.2.2 Management of Semantic Annotations.....	18
3.2.3 Management of OWL Ontologies and Ontological Entities.....	18
3.2.4 Management of SAWSDL Descriptions.....	19
3.2.5 Query Functionality	20
4 HKS Implementation.....	21
4.1 TRREE and TRREEAdapter.....	22
4.1.1 TRREE.....	22
4.1.2 TRREE Adapter.....	22
4.2 Ontology Repository Data Service	23
4.3 Storage of Documents and Text-related Annotations.....	23
4.4 Storage of Web Service Annotations	25
4.5 Querying Heterogeneous Knowledge.....	27
4.5.1 SPARQL Support.....	27
4.5.2 Full Text Search Support	27
5 Evaluation Testbed	30
5.1 Document generator.....	31
5.2 Evaluation Scenarios.....	31
6 Conclusion and Future Work	32
7 Bibliography and References	33
8 Appendix A. Schema for Representation of Semantic Annotations in RDF	34
8.1 Snippet from PROTON KM	34
8.2 SAR Encoding Schema.....	35

9	Appendix B. Quickstart Guide to the SAR Data Service	38
9.1	Repository Interfaces	38
9.2	Usage Samples	39

1 Introduction

One of the major prerequisites for the realization of the TAO transitioning methodology and infrastructure is the implementation of a common storage for all kinds of knowledge involved in the transitioning process. Such a repository shall offer the possibility to uniformly store content, semantic annotations, annotated Web services, ontologies and non-textual documents. It shall also provide support for unified query over stored knowledge in order to support semi-automatic transitioning. Given the fact that big volumes of data can be associated with a legacy application, the heterogeneous knowledge store shall feature highly scalable and efficient implementation.

A store that is capable of persisting knowledge in a unified format, as well as of querying across knowledge types can also be used for solving other enterprise problems such as: governance, unified and meaningful search over diverse content, knowledge management, etc.

In order to provide such functionality, a repository shall enable representation of different types of knowledge into a common, unified format; shall provide efficient storage, loading and querying of data; and shall be very scalable with respect to the volume of information being stored. Traditional repositories exist, that have proven to work efficiently and provide good scalability to accommodate huge volumes of data. Nevertheless, they lack dynamics needed when representing knowledge whose structure may change in time. Such knowledge for example is represented in ontologies, which are difficult to maintain in traditional databases (see [1]). These needs have led to the development of a new class of storage – semantic repositories.

Semantic repositories are typically tightly connected to a particular knowledge representation format (RDF, in most cases) and are incapable of storing data represented in different formats. Other limitations are the scalability and efficiency of storage, retrieval and querying of semantic data.

In this deliverable we present the Heterogeneous Knowledge Store, a semantic repository supporting heterogeneous knowledge management, while providing scalability and efficiency to match enterprise needs.

1.1 Relevance to TAO

An important outcome of TAO is going to be the semantic-based access to heterogeneous knowledge, structured data and metadata, that are associated with the legacy application or are result of the transitioning process. The Heterogeneous Knowledge Store (HKS) developed in WP4 is responsible for providing storage and retrieval of diverse types of data into a common semantic repository and for providing query functionality over that repository. In this section we briefly summarize what is the relevance of HKS for TAO in terms of its relation to project objectives and to the rest of the workpackages.

1.1.1 Relevance to project objectives

The Heterogeneous Knowledge Store development is directly related to the second project objective of TAO – Augmentation and integration of legacy content. It supports storage of the heterogeneous knowledge associated with the legacy application being transitioned as well as with the heterogeneous results produced

during the transitioning process. It will also be part of the TAO Suite and hence is also related to the objective of providing Transitioning Methodology and Infrastructure.

HKS is a very innovative semantic repository which advances the state of the art with respect to the ability to store, access and query different types of data in a highly scalable and efficient way. It also addresses the problem of combining semantic-based and full-text search over the semantic repository, which together with the unified storage of diverse data will enable enterprises to easily search and access all their knowledge.

1.1.2 Relation to other workpackages

HKS provides storage of all types of data that is used as input to and output of all TAO components. That is why its development was carried out in coordination with all technical workpackages. Thus, HKS is designed to meet the requirements and to easily exchange data with the tools produced in WP2 and WP3 and to be seamlessly intergrated within the TAO Suite. Additionally, TAO use cases will also use HKS in their case study implementations.

In more detail the HKS is used as a storage for the following data:

- Domain and application ontologies generated by WP2 tools;
- WP3 augmentation artefacts – semantic annotations, textual content, knowledge base instances;
- WP5 stores inputs and outputs of the transitioning process, and interacts with the HKS throughout that process, storing and retrieving intermediate results produced by the other tools.

A more detailed discussion about how TAO components use HKS and a brief analysis of their requirements can be found in Section 2.

1.2 Deliverable Outline

This deliverable is structured as follows:

- Section 2 presents a short analysis of the requirements towards HKS
- Section 3 outlines the architecture and basic design decisions
- Section4 discussed the implementationof HKS
- Section 5 describes an Evaluation Testbed that is designed for the scalability and efficiency testing of HKS, it will be used for the Evaluation carried out in the third year of the project.
- Section 6 concludes the report and identifies the functionalities to be further developed until the end of the project, as well as possible extensions outside of the project scope
- Appendices contain some additional information about the Semantic Annotation Repository data service used in the implementation of HKS

2 Heterogeneous Storage Requirements

The requirements towards the HKS data model have been discussed in [1], here we present the requirements towards the functionality of the store. These requirements are then used to define the HKS interfaces (presented in section 3.2).

In the following sections we briefly discuss identified requirements coming from different phases of the TAO transitioning process, as well as some additional requirements that we have taken into account in order to make the HKS easily exploitable.

2.1 Transitioning Process Requirements

In the context of TAO, the HKS is used as a persistent storage for all input, intermediate results and outputs of the legacy application transitioning process. As different components are involved in the transitioning process, the HKS shall support functionalities for storage of the data produced by each component. Thus, the HKS enables de-coupling of the modules, which can exchange data through the persistent store. In this section we briefly introduce the requirements that are related to each transitioning stage and the components that pose these requirements, in terms of data used as input and produced as output, and methods necessary to access and manage that data.

2.1.1 Legacy Data Uploading

The TAO transitioning process requires that a preliminary step of uploading existing artefacts of the legacy application into the TAO Suite is first carried out, see [2]. These artefacts are then used in the next phases to generate ontologies, augment content and produce semantically annotated SWS.

In order to support uploading of artefacts as well as to provide access to them in the next phases, the HKS needs to provide methods for persistent storage of any type of legacy documents, retrieval of stored documents and deleting of documents.

The TAO Suite needs to upload and classify different types of legacy documents – such as application documentation, screenshots, source code, diagrams, API documentation (e.g. Javadoc), etc. Hence, the HKS methods need to be universal to allow uniform treatment of documents, while at the same time it shall store classification information for the purposes of the TAO Suite.

2.1.2 Ontology Generation

The first task of application transitioning is to define the conceptual model of the application and its domain. This conceptualization is generated by the Ontology learning services developed in WP2 ([3]) on the basis of the legacy documents uploaded to the TAO Suite. The result is generated ontologies represented in OWL³ which are then used both for purposes of annotation of legacy documents and for semantic annotation of Web services.

According to the TAO Suite architecture, WP2 tools do not interact directly with HKS, interactions are carried out through the TAO Suite. According to [2], the following functionality for ontology management is needed:

³ <http://www.w3.org/TR/owl-features/>

- storing of OWL ontologies generated by WP2 tools,
- retrieval of stored ontologies so that they are used for Content Augmentation, for semantic annotation of Web services or for export to external ontology editors,
- updating previously stored ontologies whose content was edited by external editors,
- deletion of stored ontologies.

2.1.3 Content Augmentation

Content augmentation is the second step of the transitioning process which takes content as input and produces structured data as output. In the context of TAO, this structured data is represented in the form of semantic annotations. Semantic annotations link parts of text with classes and instances in the ontology generated in the previous step. Content augmentation can be separated into two tasks: semantic annotation – using IE some parts of the document content are marked and then linked to an ontology; and, persistent storage and lookup of augmented content, where document retrieval is based on relevance to a selected set of semantic annotations instead of relevance to words (like in keyword-based lookup). Thus, HKS plays a very important role in content augmentation as persistent storage of augmented content, providing also efficient semantic-based query and retrieval.

HKS is used by content augmentation tools not only to store resulting semantic annotations, but also in the process of content augmentation in order to access the domain ontology and to populate the knowledge base (KB) with new instances identified by IE methods. The information consolidation methods, described in D3.1, suggest that the knowledge base is populated only with consolidated instances. From there we can assume that the Content augmentation tools will internally store and manage constructed individuals and only update (populate) the KB with the final definitions. The KB is represented as an OWL ontology containing the instance data and referring to the domain ontology defining the conceptualization. Therefore, the HKS only needs to provide functionality for storage, retrieval and management of ontologies in order to provide support for knowledge-base management.

Additionally, as identified in 3.1, many content augmentation scenarios require that the user can access the content of the domain ontology, to add new instances and properties, even classes, and do this with the help of UI tools as part of the content augmentation process. In TAO user interfaces providing such functionalities will be developed in WP3 (see [4]), these user tools, as well as other content augmentation tools using the HKS outside of the project scope, will need certain support from the HKS in order to be able to carry out their work. Namely, it is required that documents and ontologies are stored together in the semantic repository and also that methods for direct manipulation (storage, loading, removal) of ontology classes, instances and properties are provided.

Legacy application content tends to be heterogeneous – including text, as well as images, video, and audio data. Content augmentation will be carried out over all types of legacy content, according to [5]. Moreover, [5] indicates that in the case of content augmentation of non-textual artefacts, the results will include a textual representation of the content, extracted from OCR and ASR tools, together with generated semantic annotations. Thus, it is required that for non-textual artefacts HKS provides persistent

storage for semantic annotations, extracted textual content and the document in the original format. The textual content stored in the HKS will enable keyword queries, while the original content can be retrieved by end users for purposes of verification of augmentation results, as well as for user-assisted semantic annotation of Web services.

All interactions between content augmentation tools and HKS will be carried out through the CA Manager (see [2]). Different manipulations of data, stored in the HKS, that the CA Manager will carry out include: retrieval of documents and ontologies, updating of the KB, saving and loading of semantic annotations, querying of stored data. Different sorts of query methods are needed for the information consolidation task of the content augmentation process, according to D3.1, some of which are: getting a KB instance by property, getting properties of a specific instance, getting all annotations related to a document, etc.

2.1.4 Semantic Annotation of Web Services

The final phase of transitioning legacy application deals with the semantic annotation of Web services in order to produce Semantic Web Services. This phase is carried out by the TAO Suite and the CA Manager with possible participation of end-users (application engineers), see [2].

In the process of semantic annotation of Web services the tools take as input existing Web service definitions of parts of the application functionality, domain ontologies generated and any semantically augmented content that is relevant. As a result Web services are annotated following the W3C SAWSDL (reference) recommendation.

In order to serve as storage for SAWSDL descriptions, HKS needs to provide functionality for (according to [2]): storing and loading SAWSDL descriptions, removal of stored descriptions from the repository and providing a list of all SAWSDL descriptions currently stored in the repository.

2.2 Requirements for General Usage

In order to develop a store which can be used also in cases not covered by the TAO scope, we have considered some external requirements.

First, the HKS should be based on an extensible architecture that will allow users (which are application programmers) to easily add support for new types of knowledge necessary for their application. This requirement was also considered when developing the unified model for representation of heterogeneous data, presented in [1].

A second requirement is related to providing interfaces for semantic annotations represented as GATE⁴ objects. This requirements can be considered both as internal and as external. This is due to the fact that, as discussed in section 2.1.3, content augmentation results are stored in HKS by the CA Manager, and not directly by the GATE components developed in WP3. On the other hand, a GATE interface will enable smooth integration of GATE components and GATE-based applications within and outside of the project scope. Such interfaces will also enable integration of HKS with other tools developed by Ontotext (such as the KIM platform) and provide a better basis for its future exploitation.

⁴ <http://gate.ac.uk/>

Finally, there is a requirement that HKS provides support for different ontology representation formats and different SWS representation frameworks. This requirement is also twofold – on the one hand, such support will enable HKS to be used by applications that need storage for semantic data that is not necessarily represented in RDF or OWL. At present, the main alternative of OWL is the Web Service Modeling Ontology⁵ (WSMO), and support for WSMO descriptions will enable usage of HKS by WSMO tools developed by Ontotext. In the future, other representation paradigms can be supported, provided that HKS has an extensible architecture as discussed above. On the other hand, the adoption of SAWSDL in TAO also ensures independence from the ontology-representation framework. Hence, the possibility to store WSMO ontologies and other descriptions will open the way to possible extensions of the TAO Suite and applications of the transitioning process to produce WSMO-compliant semantic Web services.

2.3 Conclusions

From the analysis of the different requirements towards the HKS, we can conclude that repository functionality for different types of knowledge is needed. The requirements showed that there are some common and some specific functionalities for each different kind of knowledge. To summarize:

- A repository for document storage has to provide functionality for: storage of documents, removal of documents no longer used, retrieval of associated metadata, loading documents, given an identifier. Additionally, textual content shall be made available for searching, while original content shall be kept for reference purposes.
- An ontology repository shall provide support for storage and access to whole ontologies, as well as to particular ontology entities such as concepts and instances.
- A repository for semantic annotations shall support persistence and query of annotations both in the format supported by the CA Manager, and in the format produced by GATE tools
- Semantic Web service descriptions produced by the TAO Suite shall be stored together with the rest of semantic information and content, in order to provide the user with one access point for all the information and to support queries over all the heterogenous data.

In the following section we discuss the design decisions taken and the HKS specification based on the above requirements.

⁵ <http://www.wsmo.org/>

3 HKS Design

The design of the Heterogeneous Knowledge Store considers all the requirements posed by TAO components as well as requirements for usage outside of the TAO project. In this section we outline the basic design decisions taken and briefly describe the HKS interface, which provides a common access point for any components that needs to interoperate with the store.

3.1 Functional Architecture

In accordance with the requirement analysis conclusions, we have designed the HKS architecture in such a way that different types of knowledge can be stored and managed in a common store, while specific access functionality is provided for each type of knowledge.

Figure 1, below, shows the types of data that are involved in the TAO transitioning process as well as the HKS internal components which are responsible for handling specific types of data.

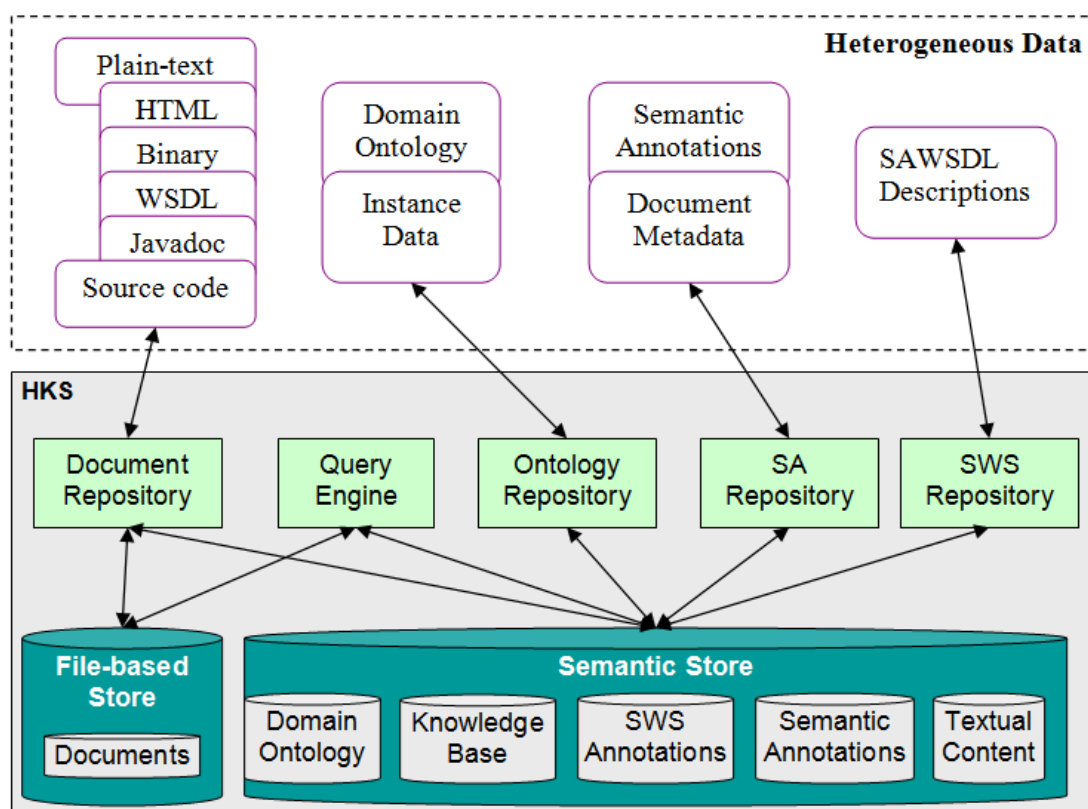


Figure 1: Functional Architecture of HKS

The main objective of HKS is to provide unified storage for heterogeneous data, so that cross document and cross knowledge type search and query can be performed. In order to achieve this, we store all data a semantic store, as illustrated by the above figure. The only exception is for legacy documents, which are also persisted in the file system (file-based store), for two reasons: first, some content cannot be represented in semantic structures, and second, to enable retrieval of documents in their original format.

As can be seen from Figure 1, our design appoints one component for each type of data – documents, ontologies, semantic annotations and semantic web services. Each component is responsible for the way in which corresponding data is written into the semantic repository and how it is read (and queried) from there.

This decision enables us to further exploit not only HKS but its subcomponents and combinations of them. As different components are relatively independent from one another, each one can be separately used in scenarios involving robust storage of particular type of knowledge. While in TAO all these components are included in HKS, there will be scenarios when only a subset of components is necessary – e.g. enterprise search not involving semantic web services. This loosely coupled design of HKS allows us to easily customize it for different domains. Furthermore, the HKS can be easily extended as components for other types of knowledge are added with no impact on the functionality of the existing ones.

In the following sections we briefly present each of the functional components of HKS.

3.1.1 Document Repository

As already discussed documents are a special kind of legacy data that needs to be stored in two places – a textual representation of the content (in the case of audio and video) is stored in the semantic repository to enable full-text search over it, while documents in the original format are kept in a file-based repository. If there is metadata provided for the document (document-level annotations), this data also needs to be stored in the semantic repository. Thus, the Document Repository is the only component that will use both the semantic and the file-based storage.

3.1.2 Ontology Repository

There are two types of tasks for ontology management – one, carried out distantly, when ontologies are modified by the user application and ontologies are persisted when this process is over, and another one, when ontology entities are directly manipulated. While methods for bulk management of ontologies are easy to use and avoid the mistakes of direct editing, they provide also efficiency in terms that all manipulations to the ontology content are carried out locally by the HKS user application. But on the other hand, methods for direct management of ontology entities are also necessary. They avoid the problems and penalties otherwise arising when manipulating big ontologies that need to be saved often (and hence require frequent communication exchanges with large data, also serialization), or when it is important for the HKS user (some application developer) that changes to the ontology are persisted while editing it, e.g. for purposes of collaborative editing or using reasoning.

We have decided that Ontology Repository shall provide both types of management functionalities, in order to be usable in different scenarios. Thus, it supports both the high-level methods needed by TAO Suite and some fine grained methods needed by content augmentation components, as listed in an appendix of [5].

As previously discussed, the OWL formalism has been selected for ontology representation in TAO. Both the ontology generation and content augmentation tools work with OWL ontologies. That is why the Ontology Repository component shall enable storage and management of OWL ontologies in the semantic repository.

Since the SAWSDL framework for semantic annotation of web services is ontology neutral, it is possible that it is used in combination with other ontology languages, such as WSMO for example. In such cases, a different implementation of Ontology Repository shall be used. Note that it will provide similar but different methods for fine grained access, since these methods are language dependent.

The Ontology Repository component is also responsible for the way in which ontologies (in a given language) are persisted in the ORDI data model. Translation from OWL into RDF is not necessary, since OWL is an extension of RDF, but for other ontology representation languages (e.g. WSMO) such translation shall be carried out. Additionally, the ORDI data model provides possibilities to associate context and triplesets with RDF statements (see [1] for more details). For ontologies represented in different formats different associations may be made, but in the common case our encodings will be such that Context refers to the ontology identifier, while different tripleset associations are made in order to encode dependencies between ontological entities. An example of such dependency, is a tripleset that associates the identifier of a class to all statements related to its subclasses. Such a dependency is then used for swift retrieval and management. Triplesets can also be used to provide user-specific access to stored data and for versioning purposes.

3.1.3 Semantic Annotations Repository

As discussed in [1], there are different types of annotations that can be produced by content augmentation tools. Content Augmentation tools in TAO produce stand-off (not embedded), semantic annotations (referring to ontologies) on two levels—document level and character level. The HKS provides support for management – storage, retrieval and deletion for both types of annotations produced in TAO.

In the context of TAO, semantic annotations are represented in RDF following the PROTON Knowledge Management (PROTON KM) schema, and its Mention and Document classes in particular (presented in [1]). Thus, semantic annotations produced by the CA Manager will be represented in RDF as instances of these classes. The complete schema for semantic annotation representation in RDF can be found in Appendix A. Semantic annotations are either passed to the Semantic Annotation Repository already serialized to RDF or the repository component is responsible for translating them to RDF (according to the adopted schema). It should be noted that the fact that annotations can be passed directly in RDF enables any schema for their representation to be used by the content augmentation component. Additionally, the Semantic Annotation Repository can receive as input GATE Documents. In this way, any content augmentation tool external to TAO can also use the HKS, because it can either pass the annotations as GATE objects, as RDF following the SAR schema, or as an RDF/XML serialization of the annotations, following a custom schema (specific for the content augmentation tool). Hence, we can say that the SAR component is schema independent by design.

When semantic annotations are stored in the semantic repository, the context element shall refer to the document for which these annotations are generated. Triplesets on the other hand, enable encoding of annotation schemas, as well as encoding document level annotations. Again, triplesets can be used to support authentication and access rights.

3.1.4 *Semantic Web Service Repository*

This functional component is responsible for the storage and management of semantic web services. In TAO, SAWSDL is adopted as the paradigm for semantic annotation of web services. Nevertheless, if other SWS framework is used in other scenarios, the HKS can be easily extended and an implementation of SWS Repository can be integrated for these purposes. For example, we provide an implementation for WSMO.

The SWS Repository shall provide methods for high level access to SWS descriptions as well as lower level access to particular elements of these descriptions. SWS descriptions are first translated into RDF, following the translation specifications for the specific framework. In the case of SAWSDL, translation is carried out following the mapping specified in ([6]). For WSMO a mapping is specified in ([7]).

In this case the context element of the ORDI data model is used to store the identifier of the SWS description, or an identifier of a document in which this description appears. Triplesets are again used to optimize management of elements of the SWS description, e.g. accessing a service capability in WSMO or a model reference in SAWSDL.

3.1.5 *Query Support*

The main advantage of being able to store heterogeneous content into one and the same semantic repository is to be able to carry out searches over it. That is why the HKS has a dedicated component that will provide a single point of querying content independent of its type. This component will support standard-based languages for querying semantic repositories. At present, there is only one such standard – SPARQL, which is hence the language supported.

In order to enable independent usage of HKS components and configurability, each of the components will also provide query interfaces.

In order to abstract end-users from the specific language used for querying, a Query Engine component will be developed that will provide high level methods for query construction. Queries will be constructed by specifying the set of variables to be involved, the ones to be present in the resultset and the restrictions over them. Queries constructed through the Query engine will then be translated into the language supported by the semantic repository and executed.

3.2 **HKS interfaces**

This section briefly describes each of the methods supported by the HKS interface. A more technical description of the methods can be found in [2].

An important decision when defining the HKS interfaces was to use Semantic Web standards as data-exchange formats – OWL, RDF, SAWSDL. Thus, we avoid using ad-hoc exchange formats and enable smooth integration of HKS in systems within and outside of the project.

It should be noted that since HKS implementation is open-source, support for additional formats (not currently implemented) can easily be added by interested parties. The HKS interface can be extended to support additional methods as long as an implementation is provided that makes the connection with ORDI. In such a way

for example, we can provide support for WSMO descriptions instead of or in addition to SAWSDL ones if an end-user scenario requires that. Currently, WSMO-related methods are not part of the HKS interface because they are not used in TAO.

3.2.1 Management of Legacy Documents

The following methods are provided to enable storage and access to legacy documents:

- *String storeDocument(String content, String docType);* - stores the content of the document in the semantic repository and returns the identifier used by HKS
- *String storeDocumentByURL(String url, String docType);* - retrieves the document from the URL provided, stores it in HKS and returns the identifier used.
- *String retrieveDocument (String docID);* – returns document content as string
- *String[] retrieveDocuments(String docType);* – returns an array with the identifiers of all documents of a given type stored in the HKS
- *void deleteDocument(String docID);* - removes the document with the given identifier from the HKS repository

3.2.2 Management of Semantic Annotations

The following methods are provided for management of annotations represented in RDF or annotations that are part of GATE Document:

- *void addAnnotations(String annotRDF, String docID);* -adds annotations given as RDF graph for the specified document. If there are more annotations in the HKS for this document, the new ones are simply added to the set.
- *void storeAnnotations(String annotRDF,String docID);* - stores given annotations, by replacing any existing annotations for the same document.
- *String retrieveAnnotations(String docID);* -returns an RDF graph, serialized to String, containing all annotations associated with the specified document identifier.
- *void deleteAnnotations(String docID);* - removes all annotations associated with the given document from the repository.
- *String storeSemanticAnnotatedDocument(Document doc);* - stores the document content and all the annotations, returns the identifier assigned to the document when stored in the HKS
- *Document retrieveDocument(String docID);* - specific method that retrieves the annotations and content stored for a document, constructs a GATE Document object and returns it.

3.2.3 Management of OWL Ontologies and Ontological Entities

The following set of methods enable management of OWL ontologies and ontological entities:

- *String addOWLontology(String ontology);* - returns the identifier with which the ontology is stored in HKS.

- *void storeOWLOntology(String ontology, String ontologyID);* - stores the OWL ontology with the specified identifier.
- *String retrieveOWLOntology(String ontologyID);* - loads previously stored OWL ontology, serialized as RDF/XML.
- *String[] listOntologies();* - gives a list of identifiers of all OWL ontologies currently stored in HKS.
- *void deleteOWLOntology(String ontologyID);* - removes an OWL ontology from HKS.
- *String addClass(String ontologyID, String parentID, String className);* - adds an OWL class with the given name, which subclasses a specified class in the ontology and returns the class identifier.
- *String addInstance(String ontologyID, String classID, String instName);* - adds an instance of a given class in the ontology and returns the identifier with which it is stored.
- *String addProperty(String ontologyID, String type, String domainID, String range);* - creates a property with the specified type, domain and range, then adds it to the ontology in HKS.
- *String getOntoResource(String ontoID, String resName);* - returns a serialized RDF graph of all statements for the given resource.
- *void removeResource(String ontologyID, String resID);* - deletes all statements for the specified ontological entity (resource).
- *String[] getEntityProperties(String ontologyID, String entityID);* - the returned array contains the set of identifiers of properties belonging to the class or inherited from its superclass, entityID shall point either to a class or to an instance.
- *void setPropertyValue(String ontologyID, String entityID, String propertyID, String value);* - sets the value of a specified property.
- *String getPropertyValue(String ontologyID, String entityID, String propertyID);* - gets the value of a specific property.
- *void removePropertyValue(String ontologyID, String entityID, String propertyID);* - deletes the value of the specified property.
- *String[] getSubClasses(String ontologyId, String classId, boolean transitive);* - returns a set of identifiers of all subclasses of a given class.
- *String[] getSuperClasses(String ontologyId, String classId:String, boolean transitive);* - returns a set of identifier of all superclasses of a given class.
- *String[] getClassInstances (String ontologyId, String classId, boolean transitive);* - returns a set of identifiers of all instances of a given class.

3.2.4 Management of SAWSDL Descriptions

This is a list of the methods currently supported by HKS for SAWSDL descriptions. Note that additional methods might be necessary and may be added in the final year of the project:

- *String storeSAWSDL(String sawsdlContent);* -returns the identifier assigned by the HKS when storing this SAWSDL description
- *String retrieveSAWSDL(String sawsdlID);* - loads the SAWSDL description given the identifier with which it was stored
- *void deleteSAWSDL(String sawsdlID);* - deletes previously stored SAWSDL definition given the document identifier.
- *String[] listSAWSDLDescriptions();* - returns an array of identifiers of all SAWSDL descriptions currently stored in the HKS.

3.2.5 Query Functionality

The HKS supports SPARQL as a query language, in order to be standard compliant. Nevertheless, the query methods are generic so other languages can easily be used if supported by the underlying semantic repository.

Currently, only one method is provided:

- *CloseableIterator query(String queryText);* - returns an iterator over the resultset.

In the future more abstract query interfaces can be added to enable users not familiar with SPARQL (or other language) to construct queries by specifying only restrictions and which variables to be included in the result.

4 HKS Implementation

The HKS is realised as a complex component based on the ORDI SG middleware. It is a specific configuration/bundle of a semantic repository, the ORDI middleware and specific data services that enable storage of heterogeneous data through translation into the ORDI data model.

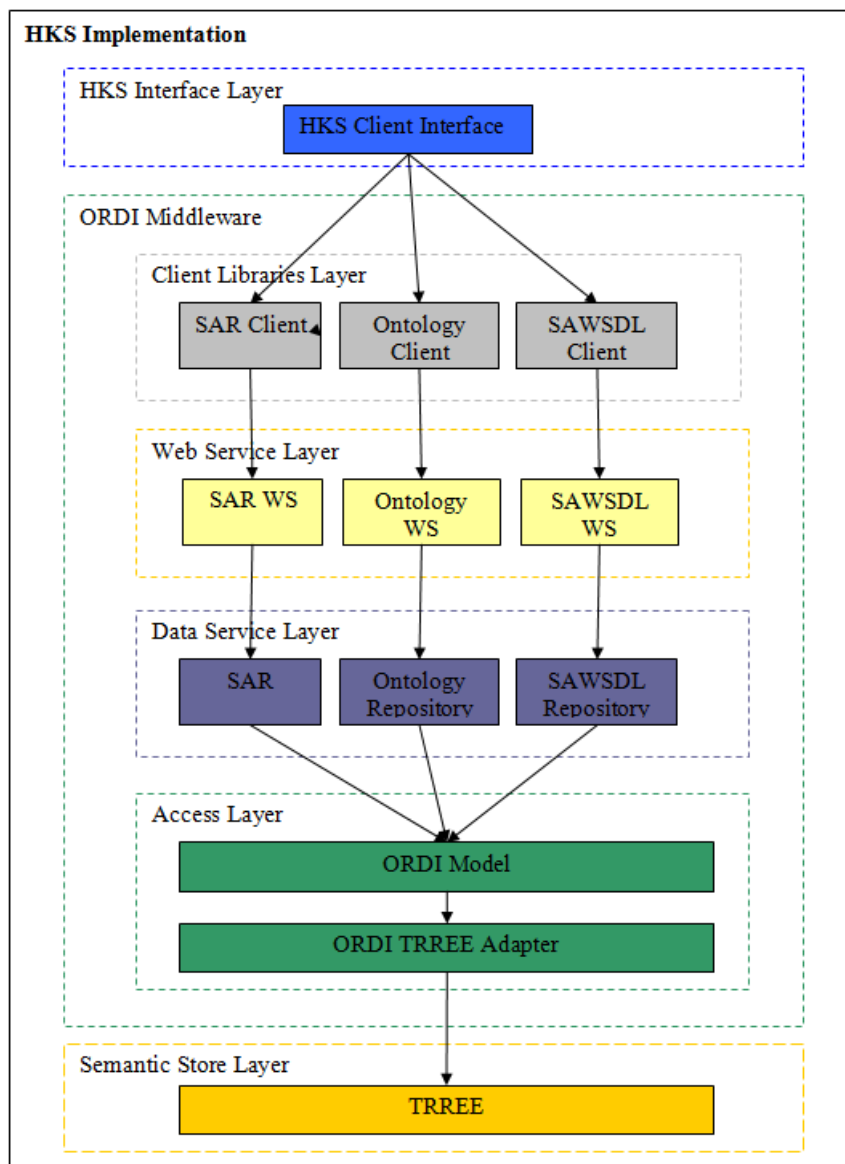


Figure 2: HKS Implementation

Figure 2 presents the implementation diagram of the HKS, whose layers and components are discussed in the rest of this section.

The Heterogeneous Knowledge Store is implemented as a layered structure. It consists of a Semantic Store Layer, ORDI Middleware comprised of different layers for data integration and data services, and a HKS Interface Layer.

The Semantic Store Layer is responsible for the persistent storage, retrieval and querying of data represented in the ORDI data model (presented in [1]). This layer is realized by TRREE. The basic internal layer of the ORDI middleware is responsible

for providing access to data sources, which in our case is the TRREE semantic repository. Over this layer, a set of data services is developed which actually supports manipulation of heterogeneous knowledge. Each data service is responsible for handling particular kind of knowledge and is realized in three layers – a basic layer, a web service and a client library for accessing the web service.

The HKS Client Interface is a thin isolation layer, created for three reasons: to provide a single point of access to all data services used in the HKS implementation, to support cross-knowledge queries and to enable authentication to be centrally carried out.

4.1 TRREE and TRREEAdapter

TRREE is a high-performance storage inference engine that integrates efficient reasoning and performs forward-chaining of entailment rules on top of RDF extended named graphs. The TRREEAdapter component implements the ORDI specification and provides a bridge between the ORDI model and the low level interfaces of the the rule engine.

4.1.1 TRREE

The HKS implementation depends on version 3.0 of TRREE. This version completely rewrites version 2.x of the engine, featuring major changes in the supported data model, operation semantics, persistence strategy and optimizations. TRREE 3.0 offers native support for extended named graphs and employs a reasoning strategy that can be described as total materialization (based on forward chaining of entailment rules). TRREE compiles entailment rules into chunks of Java code and merges the resultant code to generate the main entry point for inference.

Reasoning and query evaluation are performed in-memory. The full content of the repository is loaded into and maintained in a proprietary representation format in a computer's main memory, which makes possible very efficient retrieval and query answering.

4.1.2 TRREE Adapter

The TRREE Adapter provides a full implementation of the ORDI model and is used as reference implementation in ORDI framework 0.5.

The ORDI data model provides high-level statement repository-like interfaces while the underlying TRREE is optimized for efficiency and provides only low level functionality. The TRREEAdapter provides full support of the ORDI data model specification and extends the underlying engine in the following directions:

- **Support of RDF types:** The data model of the TRREE engine only uses integer values. This ensures very good performance in the evaluation of internal operations, but requires additional transformations when the data is represented in the ORDI data model. Custom implementation of a disk-based hash table mapping RDF types to integers is included. The adapter implements an efficient loading mechanism, which minimizes the transformation overhead.

- Lazy-loading of statements: ORDI operations are designed to provide maximum usability at reasonable efficiency. The resolving of the RDF types and retrieval of the triplesets is done only following a client request.
- Isolation of the client interaction using sessions: There is no support for connections in TRREE, which makes it difficult the memory management for the clients of the ORDI data model. The adapter implements closable connections, which deal with all associate memory resources and facilitate the resource allocation.
- Possibility to monitor newly asserted implicit statements: By default the TRREE engine works in black-box mode, because the increased performance measures. The TRREEAdapter wraps the original model and provides sophisticated capabilities to register listeners to receive notifications from the engine.

4.2 Ontology Repository Data Service

As discussed in [1], both domain conceptualizations and instance data are represented in TAO as ontologies. The HKS offers unified functionality for working with ontologies, no distinction between conceptualizations and knowledge bases is made. Ontologies in the scope of TAO can be represented in OWL, although usage of WSML is also supported by the SAWSDL. The Ontology Repository data service included in the HKS implementation supports storage and management of OWL ontologies and ontological entities. Nevertheless, HKS users that need to work with WSML ontologies can use the WSMO4RDF data service (described in more detail in section 4.4).

The Ontology Repository data service supports all the interfaces for high level management of OWL ontologies and for access to ontological entities required for the purposes of ontology generation and content augmentation tools of TAO. This data service is also responsible for encoding OWL into the ORDI data model. Since OWL is an extension of the RDF(S) syntax and the ORDI data model only defines additional elements to the RDF statements, an actual translation is not necessary.

The existing RDF parsers of Sesame⁶ are used to produce RDF statements from the OWL ontology. Then for each statement a context and triplesets are associated before storing it in ORDI. Ontology identifier or base URL is used as context of the ontology, while triplesets are used to encode relationships between classes, instances and properties.

4.3 Storage of Documents and Text-related Annotations

Storage of text-related annotations and documents is provided by the Semantic Annotation Repository (SAR) data service. It realizes both the Document Repository and Semantic Annotation Repository components of the HKS functional architecture.

Document Storage implementation is tightly coupled with the semantic annotation storage implementation. This is a consequence from the fact that for documents both metadata and content shall be stored. Furthermore, for non-textual documents (such as video and audio) being augmented by WP3 tools (see [5]) both the document original

⁶ <http://www.openrdf.org/>

content and extracted text shall be stored. Document management in TAO is carried out by TAO Suite, and happens independently of content augmentation for the purposes of registration of legacy documentation in the beginning of the TAO transitioning process. Nevertheless, providing a dedicated ORDI data service for document management is in our opinion useless, since the HKS does not aim to replace existing document repositories, but only provides functionalities relevant to content augmentation of documents.

Thus, document storage is realized within the SAR data service so it can store: document metadata, document in the original format and textual representation of content for audio and video files augmented by WP3 tools. Documents in the original format are simply stored in the file system. The storage directory/folder can be set in a configuration file by HKS users.

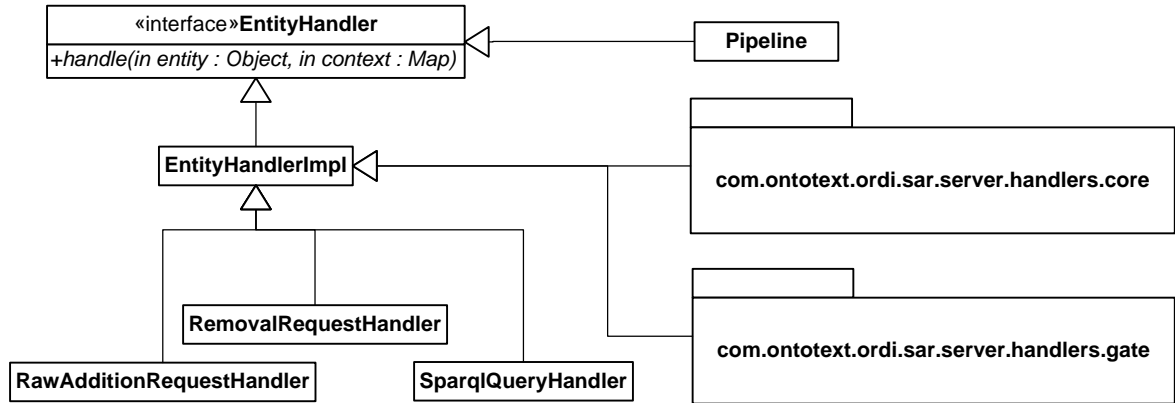
Semantic annotations in TAO are produced both on the character and on the document level. Both types of annotations can be represented in RDF following the SAR schema (see Appendix A). The SAR data service is responsible both for storage of annotations given as RDF graphs and for annotations that are represented in the GATE object model. In the case when annotations are passed to SAR serialized into RDF, the SAR data service simply adds an association of the document with all statements of the given graph. In the case when annotations are passed as GATE objects, a translation to RDF is first carried out. For GATE annotations, triplets are used to keep information about triplets and associate features to documents and annotations.

Currently, SAR provides 2 interfaces to the repository: low-level CoreRepository and high-level GateRepository.

Storage is realised as a simple pipeline (*com.ontotext.ordi.sar.server.handlers.Pipeline*). Objects supplied through the CoreRepository.store/CoreRepository.load methods are pushed inside and handled by one or more handlers as they travel along.

In the case of GATE Document, this means that different features of the object (properties, feature maps, annotation sets) are handled by a set of different handlers – they all are located in the *com.ontotext.ordi.sar.server.handlers.gate* package. The process can be recursive, i.e. an object can be re-inserted in the pipeline during its handling.

Currently, the handler objects (thus, the entities which can be handled by the pipeline) are pre-defined and cannot be modified. In future releases, however, the feature may become configurable – this will enhance greatly the spectrum of applicability of the system.



4.4 Storage of Web Service Annotations

Storage of semantically annotated web services is implemented by an SAWSDL data service. Currently it provides functionality for storage and management of complete SAWSDL descriptions, and has no methods for manipulation of SAWSDL elements. This is due to the fact that in this version of HKS, no translation from SAWSDL to RDF is implemented. Such an implementation is currently under development and will be released to TAO partners shortly. Then it will also become part of the ORDI middleware distribution (available at <http://ordi.sourceforge.net/>). When SAWSDL conversion into the ORDI data model is implemented, it will enable semantic search over the annotated web services.

The current implementation of the SAWSDL data service uses the functionality of the SAR data service. This is possible as SAWSDL descriptions can be regarded as WSDL documents to which semantic annotations can be attached. Thus, we store the WSDL document in the repository and associate with it a set of annotations (represented in the SAR schema) that specify the model reference elements and refer to the lifting and lowering schemata.

As already discussed, the SAWSDL is ontology language independent. It can be used together with RDF(S), OWL and WSMML ontologies. There are different tools supporting SAWSDL construction for the different languages, but they all produce the same SAWSDL annotations. Thus, the fact that HKS can actually store and manage ontologies represented in any of these languages, makes it usable in any scenario.

In addition, the WSMO4RDF data service makes it possible that WSMO semantic web service descriptions are persisted in the HKS. Note that the WSMO4RDF service is not included in the implementation diagram as it will not be part of the HKS implementation used in TAO. But it can be easily added and used in scenarios requiring WSMO description storage.

WSMO4RDF is the first Data Service developed within the ORDI SG Framework. It enables translation of WSMO descriptions or object models created in `wsmo4j`⁷ to the ORDI SG data model and provides an abstract repository with higher level interface enabling storage, retrieval and modification of WSMO objects instead of ORDI SG triplesets. The main functionality that is supported by WSMO4RDF is related to:

⁷ <http://wsmo4j.sourceforge.net/>

- Providing an implementation of the *WSMO Repository* interface that transparently stores, retrieves and modifies ORDI SG triplesets;
- Translation of WSMO objects into triplesets, based on [7] and using ORDI SG named graph and tripleset elements to enable versioning, provenance and easy to handle representation of WSML (addressing some of the problems of WSML to RDF translation);
- Support of a query evaluation infrastructure, that is compliant with SPARQL.

The WSMO4RDF implementation realizes the following functionality:

- WSML to RDF mapping - provides the basic representation of WSML constructs as triples (with subject, predicate and object) and ensures RDF compliance
- Uses the NamedGraph and Tripleset elements of the ORDI SG model to keep information about the source document providing the WSML definition, and the relationship between a WSMO element and the context in which it is defined.
- Default repository which implements the WSMO Repository interface, thus enabling storage and retrieval of WSMO goals, ontologies, mediators and Web service through the ORDI SG interfaces into the underlying semantic repository.

In the WSMO4RDF implementation, the NamedGraph element of the ORDI SG tripleset model is used to store a reference (IRI) to the WSMO top level element (goal, ontology, mediator or Web service) which serves as a context to the currently translated element. For example, the NamedGraph element of each quadruple describing a concept will contain the identifier of the ontology that contains this concept definition.

The Tripleset element, on the other hand, will contain references to WSMO elements of the upper levels of nested definitions. For example, the tripleset elements of all statements generated for the translation of a WSML attribute definition will contain its identifier, the identifier of the concept that attribute belongs to and the identifier of the WSML ontology it is part of. The usage of the NamedGraph and Tripleset elements has the following advantages with respect to the WSML to RDF translation:

- Efficient management of WSMO elements - due to the direct access to the related statements by the identifiers kept in the Tripleset element.
- Improved management of shared elements- there are elements that can appear as part of two or more different WSMO top-level elements (ontologies, goals, Web services, mediators). Such examples are a capability that is supported by two or more different WS or a concept that is defined in more than one ontology. In the case of a concept having definitions in two different ontologies, when one of these ontologies is removed, the concept definitions appearing in that ontology will be deleted while the definitions appearing in the other ontology will remain. The realization of shared elements management is very useful in the context of WSMO, which relies on the open-world assumption - where information is coming from different sources and ontology definitions are spread into different files.

An implementation of HKS including the WSMO4RDF service can be received on demand, or users can download the WSMO4RDF service from <http://ordi.sourceforge.net/>.

4.5 Querying Heterogeneous Knowledge

In order to be able to query the data stored in the semantic repository, a Query Engine is envisioned in the HKS architecture. It is an abstract component which provides one query interface for all data as well as high level ways of constructing queries. In the current version of HKS, this functional component is implemented as part of the HKS client interface and through query language interfaces supported by the ORDI model and TRREE.

In the future, a separate implementation of the Query Engine will be developed to enable users that are not familiar with SPARQL and the ORDI data model to easily construct queries. However such developments are outside of the TAO project scope.

In the next sections we briefly discuss the current query support provided by HKS.

4.5.1 SPARQL Support

SPARQL⁸ is the W3C standard for querying semantic repositories. HKS is designed to be standards-compliant, so we provide support for SPARQL as a query language to the semantic repository.

SPARQL support is needed on three different layers of the HKS implementation. First, it is supported by the semantic repository (TRREE) which actually performs the query evaluation and fetches the result. Then SPARQL interfaces are made available to different data services by the ORDI model (through the TRREE Adapter). Each data service also provides query interfaces while the HKS client interface provides the common query point for all used data services. The query support in the client interface actually enables cross-knowledge queries to be executed (e.g. to fetch documents containing annotations referring to concepts in an ontology which are subconcepts of Location).

SPARQL support in TRREE is implemented using the SPARQL evaluation engine provided by Sesame 2.0. Additionally, TRREE uses its indexes to optimize query evaluation and result retrieval.

4.5.2 Full Text Search Support

Sesame 2.0 does not support full text search by default. In order to match a literal in the WHERE clause of a query, a boolean expression is provided by Sesame's query evaluation model. When a literal must be matched against a string pattern, its binding is retrieved from the tuple and direct match is performed. However, this is a problem in the TRREE model, because it holds the URI and literal data on disk storage rather than keeping it in memory. This requires one IO operation on every single pattern matching which extremely slows down the overall performance of the query evaluation.

The problem can be resolved by replacing the LIKE Boolean expressions with LIKE iterators. These iterators return the internal IDs of the URIs and literals which match

⁸ <http://wsmo4j.sourceforge.net/>

the given pattern. Then the result set obtained by the LIKE iterators is intersected with the result set of the iterators over the query's triple patterns. This dramatically improves the query evaluation performance but it is still in the research stage.

Implementation difficulties of this approach are met in the process of building the full text index. We have experimented with different algorithms to test the full text indexing feature and the complexity of index building is rather different for each one of them.

The following three ways to build a full text index have been explored:

1. Prefix tree – very suitable for prefix search like “abc*” and easy to implement. However, it is not at all suitable for suffix search like “*abc”. There are some issues of this approach because of the necessity of obtaining sorted output from the iterator over the LIKE clause. That is to say, some rearrangement must be done while gathering the IDs of the URIs and literals that match the given pattern. The sorted output is required by query optimization performed by TRREE.
2. Embedding the Lucene engine to perform full text indexing. This approach is implemented and tested. The results show very fast prefix search capabilities of Lucene but the suffix search is very slow due to lack of adequate indexing. Lucene performs full scan when processing patterns of type “*abc” which is the cause for the bad performance. Furthermore, if Lucene decides that the strings with that suffix are not so many (using some statistics), it tries to rewrite the original Lucene query (that query is generated by the TRREE query optimizer out of the original SPARQL query LIKE clause). It forms an OR clause of all possible prefixes. Sometimes this OR clause may reach thousands, tens of thousands and even hundreds of thousands of items which can easily fill up the heap and crash with an OutOfMemoryError.
3. Using the suffix array algorithm. The algorithm works with sorted view of all strings (URIs and literals) and all their substrings. The strings are lexicographically sorted and this array allows for very fast prefix and suffix search (actually one can search for any part of any string because of having all suffixes already sorted). There are some issues to cope with, e.g. when pattern matching is performed, the pattern must not be matched directly with the strings of the URIs and the literals, because of the IO penalty which occurs. Some clever indexing must be performed to keep only the important information in memory and leaving all the string data on the disk storage. The whole full text index itself must be kept in-memory for performance reasons.

There is still work to do in order to get a working and efficient implementation of full text indexes. Currently we are working on the third approach because it will give much more search capabilities than the others. It must be pointed out that having an index out of which can be built an iterator over the pattern in the LIKE clause is not the only factor to accelerate the queries. The simplest case is to have only one clause in the WHERE clause – a LIKE clause (a single pattern to match).

Using the former approaches can solve it but a query can easily be complicated by adding a single NOT to the LIKE clause – the NOT LIKE clause must return the inversed result that is returned by the corresponding LIKE clause. Moreover, intersection of the iterators implicitly presumes an AND operator. Adding a single OR operator in the WHERE clause is a very big obstacle to form an iterator so additional

D4.2-Annex/ Heterogeneous Knowledge Store Software Companion

research must be carried out to identify the boolean expressions which can be converted to iterators, the boolean expressions that cannot be converted or for performance reasons are better to be left as boolean expressions.

5 Evaluation Testbed

This section describes the testbed that is developed in order to test HKS scalability and efficiency. This testbed will be used in the final year of the TAO project, and evaluation results will be reported in deliverable D4.4.

The HKS testbed is a separate module to allow the measurement of performance in terms of scalability of the storage and efficiency of loading and query. The HKS implementation consists of several, relatively independent components (ORDI Data services) and we will both measure them independently and in scenarios of combined usage.

For the purposes of testing and evaluation, all legacy and transitioning data provided by TAO use cases will be imported in HKS. In addition we also develop a generator component that will be used to generate artificial documents, so that scenarios with more documents, more entities or different entity distribution over documents could be tested and measured. This will make it possible to test the boundaries to which the HKS can scale in terms of number of documents, number of annotations and ontological entities being stored, even if the legacy applications do not involve such a big number of documents for example.

Thus, the main requirements for the testbed can be summarized as follows:

- allow for regression tests for HKS performance (and its sub components);
- allow for scalability tests (big number of documents, annotations and entities), stress- and load tests;
- provide a parameterized generator for documents and entities;
- measure the performance of the most common HKS queries used in the context of TAO.

Generation of documents and entities shall be customizable in terms of the following parameters:

- Document Generator – document-size-range as number of paragraphs, because document content is generated by paragraphs; document count (e.g. 100k); annotations-per-document-range (e.g. 10-20/paragraph).
- Entity Generator - per-class entity-count. Entities are coming either from existing ontologies (with large volume of entities) imported through the Ontology repository data service or are artificially generated by an EntityGenerator component.

In the context of TAO it is more typical that big numbers of documents and semantic annotations will be stored, as opposed to big number of ontological entities. Nevertheless, since we want to test not only the HKS but its separate components we also consider testing scenarios with huge ontologies or with a big number of smaller ontologies. In the second year of TAO we have started developing the component responsible for document and annotation scalability tests. Next we will continue with development of the ontology test component.

5.1 Document generator

Documents are generated (as textual documents) by a DocumentGenerator component, which stores in the HKS artificially generated documents. Generated documents are also internally annotated prior to storage (based on a specified ontology). The DocumentGenerator shall use the HKS interfaces for storing each different type of knowledge.

The chosen document generation strategy is to use the existing document corpus (taken from the GATE or Dassault case study), split each document to paragraphs, and then combine random paragraphs to create a document, according to a given size range. This strategy allows for efficient and scalable corpus generation.

Then we annotate the documents based on the corresponding case study ontology and store the annotated documents in the HKS using the SAR data service.

5.2 Evaluation Scenarios

The components of HKS that need to be tested are: the core engine (TRREE based), the semantic and document repository, ontology repository and SWS repository. As outlined above we will test both scalability of the components used separately, and their combined usage through HKS interfaces. In all of the cases the test scenarios are similar, as follows:

- Initially existing artefacts (e.g. coming from the use cases) are stored in the repository;
- A couple of test queries (in SPARQL) are evaluated and the response times are measured to get an average for normal conditions (when the repository is normally loaded).
- The repository is then extended with synthetic artefacts (generated documents and annotations, ontological entities). This is carried out to measure the scalability of the repository. Queries are evaluated after a certain number (e.g. 100 or 1000) of new artefacts is stored. They give us an evaluation of how the response time grows in relation to the repository size.
- Delete is also tested at regular intervals. The aim is to evaluate the speed of the delete operation which is known to be a slow operation for the core engine of HKS.

Two types of queries are used in the evaluations. The first type of query shall be relatively and shall be designed to return a fixed, small set of results, which allows for observation of the pure query evaluation time disregarding the fetch time, which could be significant (for large resultsets) and confusing (if resultset size varies). The second type of query is more complex and is designed to get big resultsets, whose size grows in linear dependency to the size of the repository and reaches tens of thousands of results. Execution of such queries will provide us with a good combined measure for query evaluation and result retrieval time. Queries containing full-text search constraints (e.g. LIKE “*xyz*”) will also be evaluated in combination with a simple structured query and with complex structured constraints, in order to evaluate the efficiency of the FTS and the average responses of combined queries.

Other scenarios may also be considered if such appear in the process of case study usage and evaluation of HKS.

6 Conclusion and Future Work

This deliverable presented the implementation of the Heterogeneous Knowledge Store. We have first discussed the requirements that have driven the HKS functionality definition, then discussed the architecture and implementation decisions.

Due to the loosely-coupled, modular architecture of HKS it provides us with different options for exploitation. First, the HKS will be exploited as part of the TAO Suite to serve as storage throughout the transitioning of legacy applications.

Additionally, data services created for HKS can be used separately or in combination to solve problems not falling within the TAO scope. One such example is the SAR data service which will be exploited as storage for purposes of knowledge management. It can also be used in combination with the Ontology Repository data service for ontology-based content augmentation tasks. Another such possibility is to combine the Ontology repository and SWS Repository data services for applications dealing with modelling and management of semantic web services.

Our plans for HKS development in the third year of the TAO project include:

- implement native storage for SAWSDL (translation from SAWSDL to RDF);
- improvements of FTS support;
- evaluation of HKS and its sub-components, which will be reported in deliverable D4.4;
- authentication functionality;
- integration with the distributed indexing and query component ;
- changes and customizations needed for the purposes of integration and usage by the two case studies.

Other enhancements of HKS outside of the TAO scope include:

- HKS configurability – to support usage of different data services in different scenarios;
- extensions for other types of knowledge – implement other data services;
- UI for administration and management of repository data.

7 Bibliography and References

- [1] – Ognyanov D., Kiryakov A., Momchev V., Velkov R., Model and Specification for Distributed Knowledge Stores, October 2006, TAO project deliverable D4.1, <http://www.tao-project.eu/resources/publicdeliverables/d4-1-final.pdf>
- [2] – Martin J., Herrero G., Capellini A., Francart T., Amardeilh F., Marinova Z., Architecture and Integration Requirements and Specifications, April 2008, TAO project deliverable D5.2, <http://www.tao-project.eu/resources/publicdeliverables/d5-2.pdf>
- [3] – Grcar M., Ontology Learning Services Library, April 2008, TAO project deliverable D2.2.2, <http://www.tao-project.eu/resources/publicdeliverables/d2-2-2.pdf>
- [4] – Marinova Z., Amardeilh F., Georgiev K., Francart T., User Tools, April 2008, TAO project deliverable D3.4.1, <http://www.tao-project.eu/resources/publicdeliverables/d3-4-1-final.pdf>
- [5] – Bontcheva K., Damljanovic D., Aswani N., Agatonovic M., Sun J., Amardeilh F., Key Concept Identification and Clustering of Similar Content, October 2007, TAO project deliverable D3.1, <http://www.tao-project.eu/resources/publicdeliverables/d3-1.pdf>
- [6] – Farrell J., Lausen H., (eds.) Semantic Annotations for WSDL and XML, August 2007, W3C Recommendation, <http://www.wsmo.org/TR/d32/v0.1/>
- [7] – Bruijn J., Kopecky J., Krummenacher R., RDF Representation of WSML, December 2006, WSML Final Draft, [6] – Farrell J., Lausen H., (eds.) Semantic Annotations for WSDL and XML, August 2007, W3C Recommendation, <http://www.w3.org/TR/sawSDL/>

8 Appendix A. Schema for Representation of Semantic Annotations in RDF

This appendix contains the schema used by the SAR data service when translating semantic annotations into RDF. It is also used by the CA Manager to represent its outcomes stored in the HKS.

The SAR schema is given here in order to enable users to construct SPARQL queries for the semantic annotations. We also include a snippet of the PROTON KM ontology which the SAR schema actually extends. The snippet contains the definition of the referred Mention class and its properties. The full PROTON KM schema can be found at: <http://proton.semanticweb.org/2005/04/protonkm>.

8.1 Snippet from PROTON KM

<snippet>

```

<rdfs:Class rdf:about="http://proton.semanticweb.org/2005/04/protons#LexicalResource"/>
<owl:Class rdf:ID="Mention">
  <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
  >Mention</rdfs:label>
  <rdfs:subClassOf rdf:resource="http://proton.semanticweb.org/2005/04/protons#LexicalResource"/>
</owl:Class>
<owl:ObjectProperty rdf:ID="refersInstance">
  <rdfs:domain rdf:resource="#Mention"/>
  <rdfs:range rdf:resource="http://proton.semanticweb.org/2005/04/protons#Entity"/>
  <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
  >refersInstance</rdfs:label>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="occursIn">
  <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string" >occursIn</rdfs:label>
  <rdfs:range rdf:resource="http://proton.semanticweb.org/2005/04/proton#InformationResource"/>
  <rdfs:domain rdf:resource="#Mention"/>
</owl:ObjectProperty>
<owl:DatatypeProperty rdf:ID="hasEndOffset">
  <rdfs:domain rdf:resource="#Mention"/>
  <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
  >hasEndOffset</rdfs:label>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="hasStartOffset">
  <rdfs:domain rdf:resource="#Mention"/>
  <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
  >hasStartOffset</rdfs:label>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="hasString">
  <rdfs:domain rdf:resource="#Mention"/>
  <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string">hasString</rdfs:label>
</owl:DatatypeProperty>

```

</snippet>

8.2 SAR Encoding Schema

```

<rdf:RDF
  xmlns="http://www.ontotext.com/ordi/sar#"
  xmlns:protege="http://protege.stanford.edu/plugins/owl/protege#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:protonkm="http://proton.semanticweb.org/2005/04/protonkm#"
  xml:base="http://www.ontotext.com/ordi/sar">
  <owl:Ontology rdf:about="">
    <owl:imports rdf:resource="http://proton.semanticweb.org/2005/04/protonkm"/>
  </owl:Ontology>
  <owl:Class rdf:ID="FeatureMap"/>
  <owl:Class rdf:ID="FeatureMapAsProps">
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >Features are stored as properties of the FeatureMap instance</rdfs:comment>
    <rdfs:subClassOf rdf:resource="#FeatureMap"/>
  </owl:Class>
  <owl:Class rdf:ID="FeatureMapAsBag">
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <rdf:Description rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#type">
            <rdfs:domain>
              <owl:Class>
                <owl:unionOf rdf:parseType="Collection">
                  <rdf:Description rdf:about="http://www.w3.org/2002/07/owl#Thing"/>
                  <owl:Class rdf:ID="GateDocument"/>
                  <owl:Class rdf:about="#FeatureMapAsBag"/>
                  <owl:Class rdf:ID="Feature"/>
                </owl:unionOf>
              </owl:Class>
            </rdfs:domain>
          </rdf:Description>
        </owl:onProperty>
        <owl:hasValue rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >rdf:bag</owl:hasValue>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:hasValue rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
      >ordi_sa:GateDocument</owl:hasValue>
      <owl:onProperty rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#type"/>
    </owl:Restriction>
  </rdfs:subClassOf>

```

D4.2-Annex/ Heterogeneous Knowledge Store Software Companion

```
</rdfs:subClassOf>
<rdfs:subClassOf rdf:resource="#FeatureMap"/>
<rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
>Features are stored as rdf:bag</rdfs:comment>
</owl:Class>
<owl:Class rdf:about="#GateDocument">
  <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
  >The textual content of the document as a (big) literal</rdfs:comment>
</owl:Class>
<owl:Class rdf:about="#Feature">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:hasValue rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
      >rdf:li</owl:hasValue>
      <owl:onProperty rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#type"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:hasValue rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
      >ordi_sa:GateDocument</owl:hasValue>
      <owl:onProperty rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#type"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
</owl:Class>
<owl:Class rdf:ID="AnnotationSet"/>
<owl:DatatypeProperty rdf:ID="hasAnnotation">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
  >ordi_sa:hasAnnotation</rdfs:label>
  <rdfs:domain rdf:resource="#AnnotationSet"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="hasAnnotationSet">
  <rdfs:domain rdf:resource="#GateDocument"/>
  <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
  >ordi_sa:hasAnnotationSet</rdfs:label>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>
<owl:FunctionalProperty rdf:ID="hasFeatureMap">
  <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
  >ordi_sa:hasFeatureMap</rdfs:label>
  <rdfs:domain rdf:resource="#GateDocument"/>
  <rdfs:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="hasValue">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
```

D4.2-Annex/ Heterogeneous Knowledge Store Software Companion

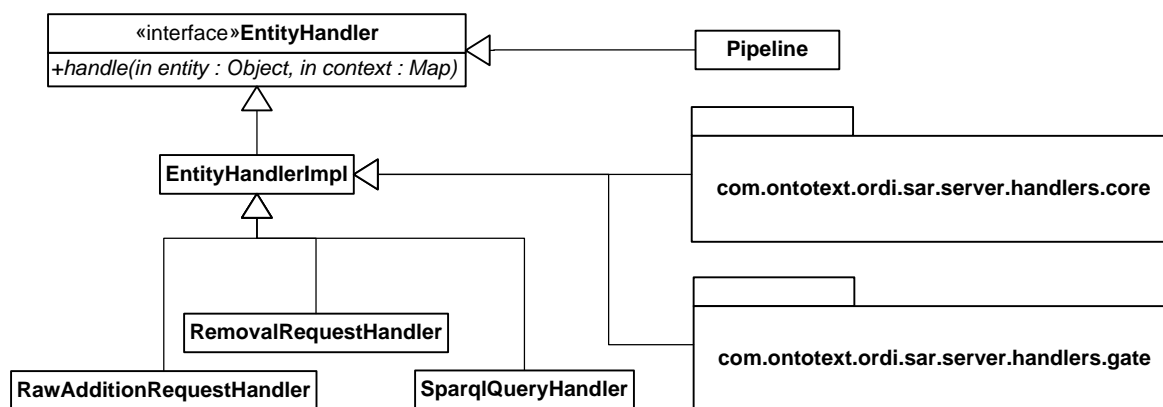
```
<rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
>ordi_sa:hasValue</rdfs:label>
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
<rdfs:domain rdf:resource="#Feature"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="hasContent">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
  <rdfs:domain rdf:resource="#GateDocument"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
  >ordi_sa:hasContent</rdfs:label>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="annotationType">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
  <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
  >ordi_sa:annotationType</rdfs:label>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="hasKey">
  <rdfs:domain rdf:resource="#Feature"/>
  <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
  >ordi_sa:hasKey</rdfs:label>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
</owl:FunctionalProperty>
</rdf:RDF>
```

9 Appendix B. Quickstart Guide to the SAR Data Service

SAR repository storage is realised as a simple pipeline (*com.ontotext.ordi.sar.server.handlers.Pipeline*). Objects supplied through the *CoreRepository.store* / *CoreRepository.load* methods are pushed inside and handled by one or more handlers as they travel along.

In the case of GATE Document, this means that different features of the object (properties, feature maps, annotation sets) are handled by a set of different handlers – they all are located in the *com.ontotext.ordi.sar.server.handlers.gate* package. The process can be recursive, i.e. an object can be re-inserted in the pipeline during its handling.

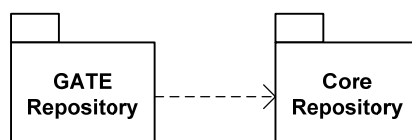
Currently, the handler objects (thus, the entities which can be handled by the pipeline) are pre-defined and cannot be modified. In future releases, however, the feature may become configurable – this will enhance greatly the spectrum of applicability of the system.



9.1 Repository Interfaces

Currently, SAR provides 2 interfaces to the repository: *CoreRepository* and *GateRepository*.

CoreRepository – this is the low-level interface. It is very close to the *EntityHandler*’s signature – the replacement of the single “handle” method with the more specific “store” and “load” methods aims to narrow the contract to more repository – oriented functionality. Operations like querying or removal are realized by supplying request objects (*com.ontotext.ordi.sar.core.requests*) to the respective method.



GateRepository – a step higher in terms of specialization, this interface is to be used to interact with the GATE-oriented part of repository in a more simplistic fashion. **GateRepository** is more or less a simple wrapper around **CoreRepository**. It is added for convenience mainly.

9.2 Usage Samples

Storing a GATE document

```
Document doc;
// document is initialized here...
GateRepositoryClient client;
GateRepositoryConnection conn;
//...
client = new GateRepositoryClient();
conn = client.getRepositoryConnection();
// ...
conn.store(doc, null);
```

Storing a single annotation

```
Document doc;
// document is initialized here...
// pick an annotation
AnnotationSet set = super.gateDoc.getAnnotations();
Annotation annotation = set.get(1100);
// connect to the repository
GateRepositoryClient client = new GateRepositoryClient();
GateRepositoryConnection conn = client.getRepositoryConnection();

String docUri = NamingUtility.documentUri(doc);
// shoot...
this.conn.store(annotation, docURI, "ann set", null);
```

Storing raw data

```
public static final String toRdfXml(Annotation annot, String setName, String ng) {
    StringWriter strWriter = new StringWriter();
    RDFXMLPrettyWriter writer = new AnnotationRdfWriter(strWriter);

    TFactory factory = new TFactoryImpl();
    try {
        writer.startRDF();
        URI namedGraph = factory.createURI(ng);
        String setUri = NamingUtility.annotationSetUri(ng, setName);
        String annotationUri = NamingUtility.annotationUri(setUri, annot);

        // store statements related to the annotation set
        // set    <hasAnnotation>    uri
        URI setUriObj = factory.createURI(setUri);

        Resource subj = setUriObj;
        URI pred = factory.createURI(NamingUtility.HAS_ANNOTATION);
        Value obj = factory.createLiteral(annotationUri);

        Statement st = factory.createStatement(subj, pred, obj, namedGraph,
setUriObj);
        writer.handleStatement(st);
        subj = factory.createURI(annotationUri);
        // hasStartOffset..
        pred = factory.createURI(NamingUtility.HAS_START_OFFSET);
        obj = factory.createLiteral(annot.getStartNode().getOffset());
```

```

        st = factory.createStatement(subj, pred, obj, namedGraph, setUriObj);
        writer.handleStatement(st);
        // hasEndOffset..
        pred = factory.createURI(NamingUtility.HAS_END_OFFSET);
        obj = factory.createLiteral(annot.getEndNode().getOffset());
        st = factory.createStatement(subj, pred, obj, namedGraph, setUriObj);
        writer.handleStatement(st);
        // annotationType
        pred = factory.createURI(NamingUtility.ANNOTATION_TYPE);
        obj = factory.createLiteral(NamingUtility.escape(annot.getType()));
        st = factory.createStatement(subj, pred, obj, namedGraph, setUriObj);
        writer.handleStatement(st);
        writer.endRDF();
    } catch (RDFHandlerException e) {
        throw new RuntimeException(e);
    }
    strWriter.flush();
    String rdf = strWriter.getBuffer().toString();
    return rdf;
}
//.....
CoreRepositoryClient client = new CoreRepositoryClient();
CoreRepositoryConnection conn = client.getRepositoryConnection();
// ...
String rdfXml = toRdfXml(annotation, setName, documentUri);

try {
    conn.storeRaw(rdfXml, "rdfxml", documentUri);
}
catch(SARException e) {
    // do something with the error
}
finally {
    conn.close();
}
}

```

Deleting a document

```

Document doc;
// document is initialized here...
CoreRepositoryClient client = new CoreRepositoryClient();
CoreRepositoryConnection conn = client.getRepositoryConnection();
String docId = NamingUtility.documentUri(doc);
try {
    conn.store(doc, null);
    // ...
    conn.delete(docId);
}
catch(SARException e) {
    // do something with the error
}
finally {
    conn.close();
}
}

```

Querying

```

CoreRepositoryClient client = new CoreRepositoryClient();
CoreRepositoryConnection conn = client.getRepositoryConnection();
conn.store(doc, null);
String docId = NamingUtility.documentUri(doc);
String sparql = String.format("SELECT ?pred ?obj"
                               + "\n { "
                               + "\n   <%s> ?pred ?obj."
                               + "\n }"
                               , docId,
NamingUtility.HAS_ANNOTATION_SET);
try {
CloseableIterator<? extends BindingSet> result = conn.query(sparql);
}
catch(SARException e) {
// do something with the error
}
finally {
conn.close();
}
// print results
while(result.hasNext()){
BindingSet set = result.next();
System.out.println(set);
}

```