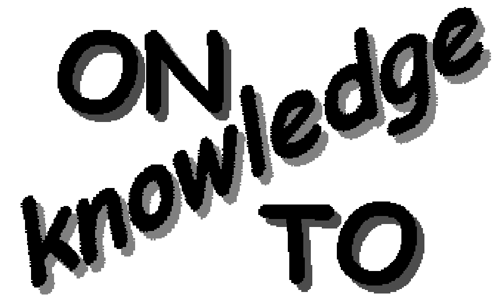


OMM Integration

Atanas Kiryakov, Borislav Popov, Damyan Ognyanov

OntoText Lab.

Identifier	**41**
Class	**Deliverable**
Version	**1.0**
Version date	**Spetember 2002**
Status	**Final**
Distribution	**Public**
Responsible Partner	**OntoText Lab., Sirma AI Ltd.**

On-To-Knowledge Consortium

This document is part of a research project funded by the IST Programme of the Commission of the European Communities as project number IST-1999-10132. The partners in this project are: Vrije Universiteit Amsterdam (VU) (co-ordinator), NL; the University of Searchrlsruhe, Germany; Schweizerische Lebensversicherungs- und Rentenanstalt / Swiss Life, Switzerland; British Telecommunications plc, UK; CognIT a.s, Norway; EnerSearch AB, Sweden; AIdministrator Nederland BV, NL; OntoText Lab., Sirma AI EOOD, Bulgaria.

Vrije Universiteit Amsterdam (VU)
Faculty of Sciences, Division of Mathematics and Computer Science
De Boelelaan 1081a
1081 HV Amsterdam, the Netherlands
Fax and Answering machine: +31-(0)20-872 27 22
Mobil phone: +31-(0)6-51850619
Contactperson: Dieter Fensel
E-mail: dieter@cs.vu.nl

University of Karlsruhe
Institute AIFB
Searchiserstr. 12
D-76128 Searchrlsruhe, Germany
Tel: +49-721-608392
Fax: +49-721-693717
Contactperson: R. Studer
E-mail: studer@aifb.uni-searchrlsruhe.de

Schweizerische Lebensversicherungs- und Rentenanstalt / Swiss Life
Swiss Life Information Systems Research Group
General Guisan-Quai 40
8022 Zürich, Switzerland
Tel: (41 1) 284 4061, Fax: (41 1)284 6913
Contactperson: Ulrich Reimer
E-mail: Ulrich.Reimer@swisslife.ch

British Telecommunications plc
BT Adastral Park
Martlesham Heath
IP5 3RE Ipswich, UK
Tel: (44 1473)605536, Fax: (44 1473)642459
Contactperson: John Davies
E-mail: John.nj.Davies@bt.com

CognIT a.s
Busterudgt 1.
N-1754 Halden, Norway
Tel: +47 69 1770 44, Fax: +47 669 006 12
Contactperson: Bernt. A. Bremdal
E-mail: bernt@cognit.no

EnerSearch AB
SE 205 09 Malmö, Sweden
Tel: +46 40 25 58 25; Fax: +46 40 611 51 84
Contactperson: Hans Ottosson
E-mail: hans.ottosson@enersearch.se

AIdministrator Nederland BV
Julianaplein 14B
3817CS Amersfoort, NL
Tel: (31-33)4659987, Fax: (31-33)4659987
Contactperson: Jos van der Meer
E-mail: Jos.van.der.Meer@aidministrator.nl

OntoText Lab.
Sirma AI EOOD - Artificial Intelligence Labs
38A Chr. Botev blvd, 1000 Sofia, Bulgaria
Tel: +359 2 981 23 38; Fax: +359 2 981 90 58
Contactperson: Atanas Kiryakov
E-mail: Atanas.Kiryakov@sirma.bg

Abstract

The Ontology Middleware Module (OMM[1]) is an extension of the Sesame[2] RDF(S) repository that adds tracking changes, meta-information, fine-grained access control, and multi-protocol client access (RMI, SOAP.) This document presents number of issues related to usage and integration of OMM and hence of the Sesame repository which OMM extends. It covers four major issues: interoperation with OMM/Sesame through different protocols; integration with OntoEdit; integration of BOR[3] reasoner with Sesame; experience with the Enersearch Case Study. Taken together with the analysis and design presented in Deliverable 38, [Kiryakov et al, 2002], it provides a full roadmap for building enterprise knowledge management system based on robust RDF(S) repository and Knowledge Control System.

Acknowledgements

The research reported here was carried out in the course of the On-To-Knowledge project. This project is partially funded by the IST Programme of the Commission of the European Communities as project number IST-1999-10132. The partners in this project are: Vrije Universiteit Amsterdam VUA (coordinator, NL), University of Karlsruhe (Germany), Swiss Life (Switzerland), BT plc (UK), CognIT a.s. (Norway), EnerSearch AB (Sweden), AIdministrator Nederland BV (NL), OntoText Lab. (BG). We wish to particularly thank to the Arjohn Kampman, Jeen Broekstra for the instant support and most specially for the timely implementation of the truth maintenance system of Sesame.

[1] See http://www.ontotext.com/omm for more information and downloads. A publicly available OMM server can be found at http://omm.ontotext.com/

[2] See http://sesame.aidministrator.nl/ and Deliverable 9 "*Sesame: A generic Architecture for Storing and Querying RDF and RDF Schema*", [Broekstra and Kampman, 2001b]

[3] See http://www.ontotext.com/bor and Deliverable 39 "*BOR: a Pragmatic DAML+OIL Reasoner*", [Simov and Jordanov, 2002]

Contents

1 Introduction ... 6
1.1 Ontology Middleware Overview ... 6
1.2 Architecture and Interfaces ... 7
1.2.1 Overview of the SESAME Architecture ... 8
1.2.2 How OMM Fits in the Picture? ... 8
1.3 Integration Overview ... 10
1.3.1 Extensions to the Sesame Web Interface ... 10
2 Integration and Remote Access ... 12
2.1 The OMM API, Reference and Public Server ... 12
2.1.1 Correspondence with the Sesame's Java library for HTTP Clients ... 13
2.1.2 OMM API Reference ... 13
2.1.3 OMM Public Server ... 16
2.2 Built-in Usage of OMM and Sesame ... 17
2.2.1 Dependencies ... 17
2.2.2 Initialization ... 17
2.2.3 Access APIs ... 18
2.3 RMI Access ... 19
2.3.1 Client ... 19
2.4 SOAP Access ... 20
2.4.1 Server ... 20
2.4.2 Client ... 20
2.5 Comaprative Protocol Performance Analysis ... 21
3 Integration with OntoEdit ... 22
3.1 Availability ... 22
3.2 Quick User's Guide ... 22
4 Integration of BOR reasoner with Sesame ... 26
4.1 Motivation ... 26
4.2 Design Overview ... 26
4.3 Usage Scenario ... 27
4.4 User interface and demo ... 28
4.5 Known Issues ... 28
4.5.1 Inconsistency: Literal sameClassAs Literal in DAML+OIL schema ... 28

4.5.2 Why we need separate properties for definitions and qualifications? 30

4.6 Improvement Plans 31

5 Experience with the Enersearch Case Study 31

5.1 Alignment of the RDF(S) File Structuring 31

5.2 Performance Observations 32

5.2.1 In-Memory SAIL 32

5.3 Lessons Learned 33

6 Conclusion 33

7 References 33

1 Introduction

The Ontology Middleware Module (OMM[4]) is an extension of the Sesame[5] RDF(S) repository that adds tracking changes, meta-information, fine-grained access control, and multi-protocol client access (RMI, SOAP.) This document presents number of issues related to usage and integration of OMM and hence of the Sesame repository which OMM extends. It covers four major issues: interoperation with OMM/Sesame through different protocols; integration with OntoEdit; integration of BOR[6] reasoner with Sesame; experience with the Enersearch Case Study. Taken together with the analysis and design presented in Deliverable 38, [Kiryakov et al, 2002], it provides a full roadmap for building enterprise knowledge management system based on robust RDF(S) repository and Knowledge Control System.

In order to enable better understanding of the OMM nature, the rest of this section provides overview of the definition and design of OMM followed by presentation of the architecture on a technical level and overview on the various integration aspects. The second section provides extensive guide for interoperation with OMM through different access protocols. Technical details and lessons from the OMM integration with OntoEdit are reported in the next section. The fourth section presents the approach for integration of the BOR DAML+OIL reasoner with sesame. Section five is dedicated to the experience gathered with the Enersearch case study (ref… Del27, Del28). Finally, the last section provides a short conclusion.

Note: This documents contains parts from the OMM System Documentation [OntoText, 2002].

1.1 Ontology Middleware Overview

The middleware can be seen as „administrative“ software infrastructure that makes the results of the On-To-Knowledge project easier for integration in real-world applications. The central issue is to make the methodology and modules available to the society in a shape that allows easier development, management, maintenance, and use of middle-size and big knowledge bases[7]. In the light of these objectives the following main features are supported:

- Versioning (tracking changes) of knowledge bases;
- Access control (security) system;
- Meta-information for knowledge bases.

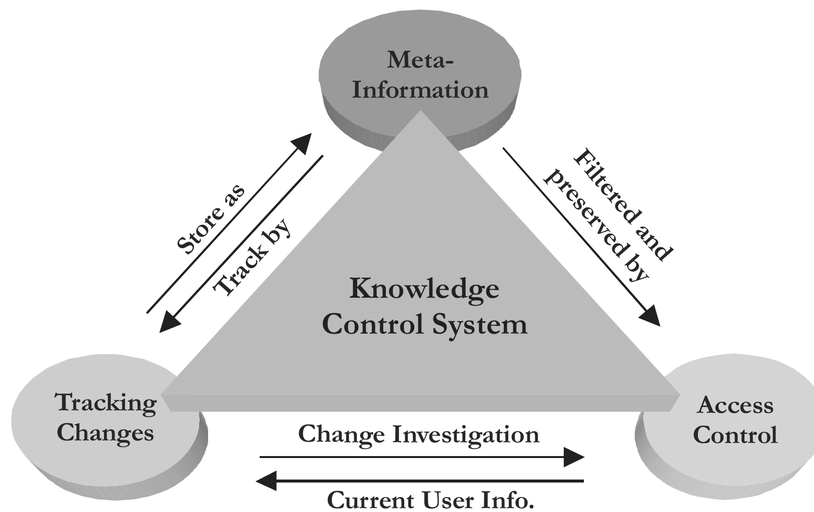
These three aspects are tightly interrelated among each other as depicted on the following scheme:

[4] See http://www.ontotext.com/omm for more information and downloads. A publicly available OMM server can be found at http://omm.ontotext.com/

[5] See http://sesame.aidministrator.nl/ and Deliverable 9 "*Sesame: A generic Architecture for Storing and Querying RDF and RDF Schema*", [Broekstra and Kampman, 2001b]

[6] See http://www.ontotext.com/bor and Deliverable 39 "*BOR: a Pragmatic DAML+OIL Reasoner*", [Simov and Jordanov, 2002]

[7] A knowledge base can consist of ontology and/or instance data and application specific knowledge.



The composition of the three functions above represents a *Knowledge Control System* (KCS) that provides the knowledge engineers with the same level of control and manageability of the knowledge in the process of its development and maintenance as the source control systems (such as CVS) provide for the software. However, KCS is not only limited to support the knowledge engineers or developers – from the perspective of the end-user applications, KCS can be seen as equivalent to the database security, change tracking (often called cataloguing) and auditing systems. The KCS is be carefully designed so to support these two distinct use cases.

A fully-functional O*ntology Middleware* system should serve as a flexible and extendable platform for knowledge management solutions. It has to provide infrastructure with at least the following features:

- A repository providing the basic storage services in a scalable and reliable fashion. This role is already fulfilled by SEASME.
- Knowledge control – the KCS introduced above.
- Multi-protocol client access to allow different users and applications to use the system via the most efficient "transportation" media. This aspect is discussed in the next sub-section.
- Support for plugable reasoning modules suitable for various domains and applications. This ensures that within a single enterprise or computing environment one and the same system may be used for various purposes (that require different reasoning services and expressivity) so enabling easy integration, interoperability between applications, knowledge maintenance and reuse.

The design of the ontology middleware module presented here is just an extension of the SESAME architecture (see [Broekstra and Kampman, 2001b]).

1.2 Architecture and Interfaces

Ontology Middleware Module (OMM) is designed to extend the existing functionality of the SESAME RDF(S) repository enriching it with support for versioning (tracking changes), meta-information, access control (security), and DL reasoning.

Here we present an overview of the enhancements to the Sesame – how the architecture was extended, which are the new and the modified modules.

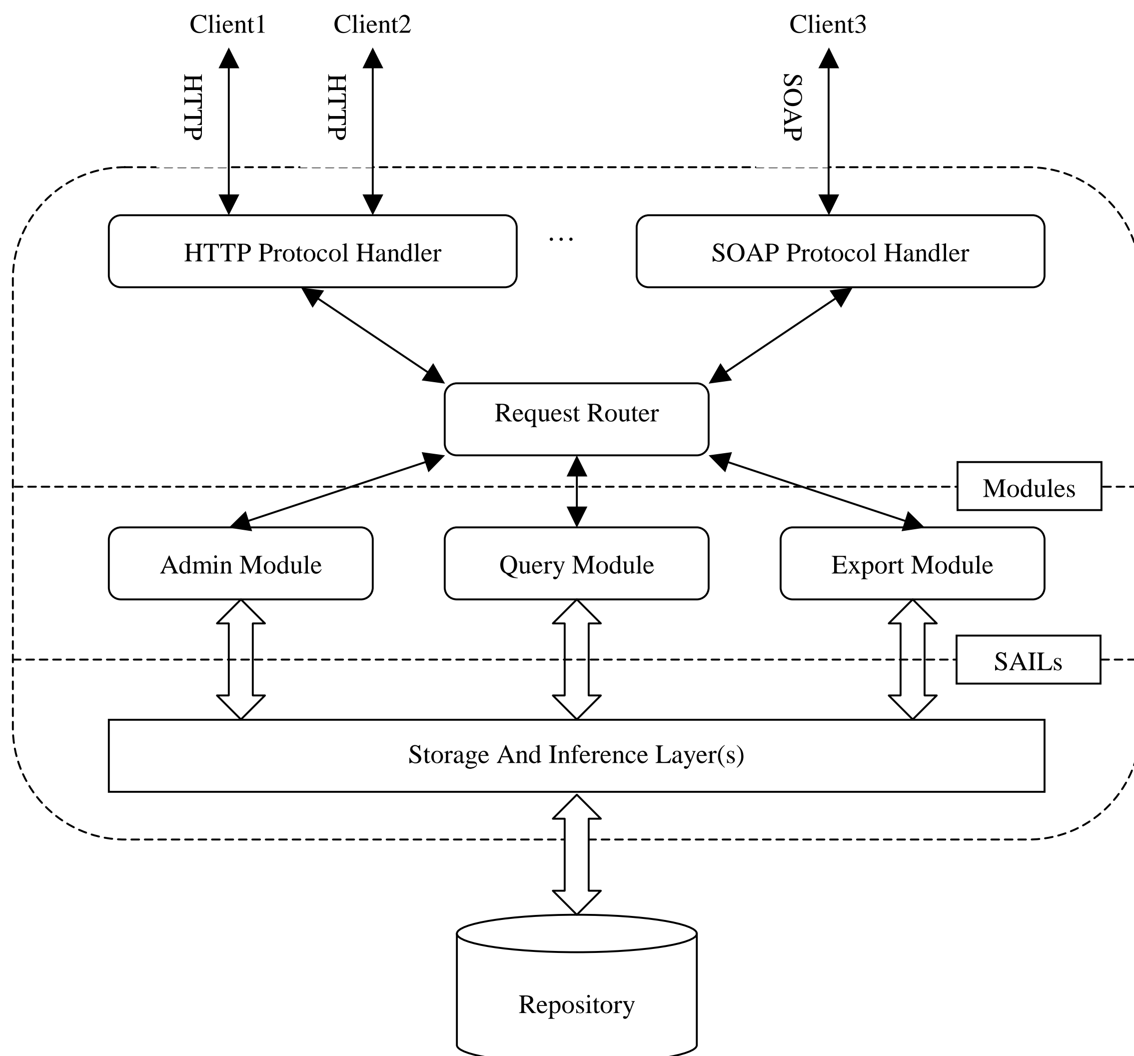
1.2.1 Overview of the SESAME Architecture

The SESAME architecture is composed of several layers. The access layer is responsible to provide all the necessary functionality managing the relations with the users, i.e. the front-end applications.

The Request Router manages the necessary marshalling of the calls from and to the appropriate protocols and routes the calls to the modules that provide the appropriate functionality – it was extended to meet the multi-protocol access requirements of OMM as well as to provide session handling in a manner desired by the versioning and security features of OMM.

Follows a layer consisting of various modules each of which implements the basic functionality via calls to the storage and inference layers (SAIL).

The SAIL interface hides the specific implementation dependant to the underlying physical storage, features and formal semantics supported. A number of SAILs can be stacked on top of each other, each of them handling some of calls and transmitting the rest to the underlying one. The access to the SAILs as well as the communication between goes through the SAIL interfaces. More about the SESAME architecture and features can be found in [Broekstra and Kampman, 2001b].



1.2.2 How OMM Fits in the Picture?

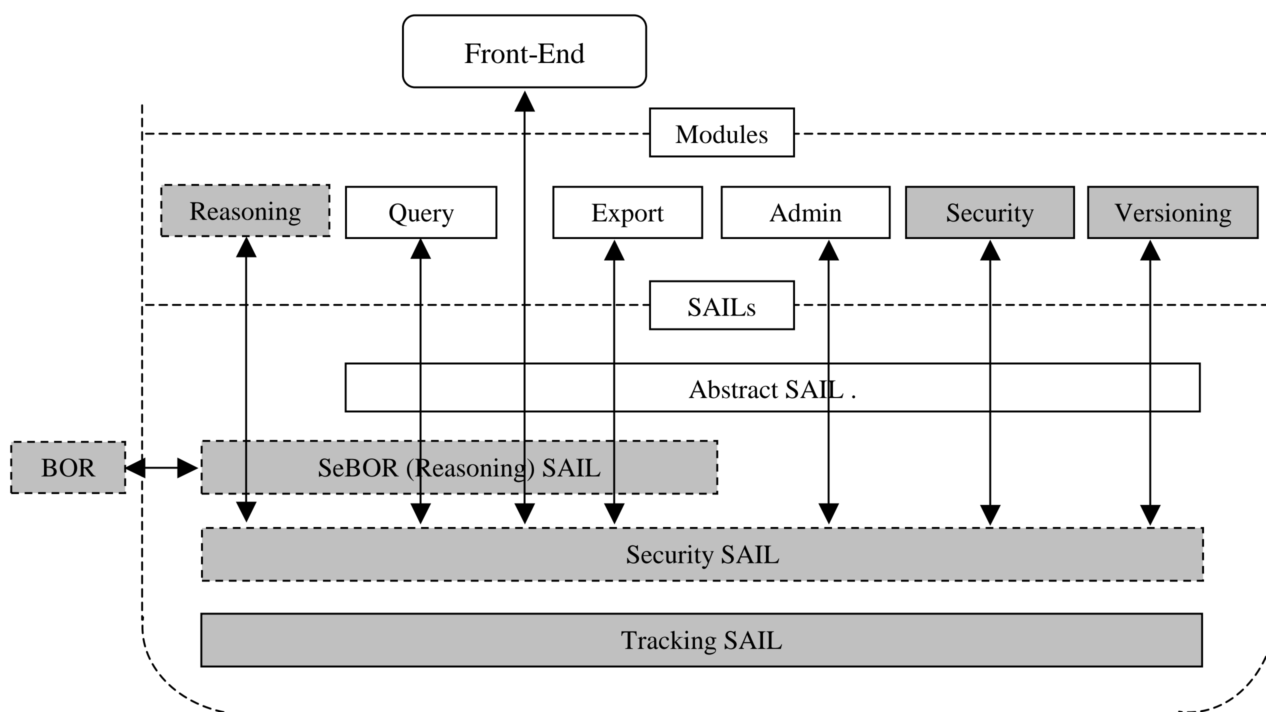
The interfaces of the OMM are designed to fit in the stacked SAIL architecture by extending it with their specific functionality. Some of the SESAME functional modules were modified so to

benefit directly from the interfaces in the OMM SAIL extension. The implementation of BOR (the DAML+OIL reasoner, [Simov and Jordanov, 2002]) was integrated through an additional SAIL. Its full potential is accessible via a separate Reasoning module responsible for routing the appropriate calls to the reasoner. On the other hand the reasoner supports the SAIL interface so the rest of the modules can also interoperate with it like with the standard RDF(S) supporting SAILs. For instance, the Query module is able to perform queries against DAML+OIL repositories without changes to its interfaces.

It is also the case that the existing basic SAIL (that implements the physical storage and the RDF(S) reasoning on top of a relational database) have been modified so to take care at the lowest level for some of the kinds of meta-information, especially those necessary for the tracking of changes. This was achieved with sub-classing of the appropriate basic Sesame SAIL. The security sub-system is implemented as a separate SAIL which may or may not be used for specific repository.

This architecture allows transparent manipulation of the repository for the existing tools. Each application can either work with the repository through the existing SESAME modules or change the repository behavior working directly with the new reasoning, versioning, and security modules or gaining the access to the underlying low-level (SAIL) programming interfaces.

The scheme below represents how the lower part of the current SESAME architecture was changed. The Request Router also undertook some modifications so to be able to (i) route the new interfaces, (ii) support more protocols, and (iii) handle the user sessions in a more comprehensive fashion. There were also important changes that take place in the router in order to enable the most convenient access control mechanism for each of the specific protocols.



The gray boxes denote the new modules or extensions developed. The arrows depict how the requests of the different modules are passing through the different SAILs. The boxes with dashed borders (Security, Reasoning and BOR) are optional, i.e. not an obligatory part of the OMM/SESAME architecture.

Another new feature is that front-end applications (such as editors and viewers) are now able to communicate directly with the SAILs without using any of the functional modules. The requests of such application (as all the requests coming outside SESAME) are handled through the request

router. It also means that such requests are subject of the standard access control and they can be made through all the supported protocols (but HTTP at present).

1.3 Integration Overview

There are two major tools OMM has to integrate within the OTK toolset – Sesame and OntoEdit. OMM is integrated with Sesame to extend that it is just its extension. The OntoEdit integration was necessary because it is the only natural general-purpose front-end tool within the suite, an editor that allows easy visualization and management of ontologies and instance knowledge. At the interface level, the rest of the tools can make use of the existing HTTP access protocol for connection to Sesame while in the same time benefiting from the Knowledge Control System (KCS) functionality – tracking changes, security, and meta-information. It is the case that all the KCS functionality was designed and implemented in such a way, that the major benefits are available transparently.

For instance, if a Sesame/OMM repository was set up with a tracking SAIL, the updates in the repository will be tracked doesn't matter what access protocol is used. Even if the tool that is making the updates is not aware of the versioning specific methods, the history of the repository can be seen and managed through other tools aware of those or through the standard Web interface.

Analogously, if there is a security SAIL in the stack of the repository, the access will be controlled with respect to the security policy defined without difference of the access method used. As it should be expected, the fact that a user or application is not aware of OMM and the security sub-system will not allow him to perform actions outside this permissions. What the user is not allowed to see is being filtered, so, he receives only the information he is expected to in a completely transparent way. In case of attempt for adding/uploading knowledge without the appropriate permissions, the non-authorized portions are being ignored and the appropriate warnings are issued.

The OntoEdit integration is addressed in section 3. A reference of the OMM API together with the instructions for usage of the various integration options is provided in the next section. To assure the compatibility with the rest of the tools we experimented with the tools and data involved in the Enersearch case study – the results of this experiment are presented in section 6. Immediately below, a short presentation of the updates in the Sesame web interface is given.

Of course, BOR reasoner developed under the project by OntoText Lab, was also integrated with OMM/Sesame as it was designed. The integration approach taken allows much of the DAML+OIL semantics to be used as easy as it happens with regular RDF(S) repositories. This way all the tools integrated with Sesame, can take benefit of using a more powerful description-logic based formalism through the very same interfaces. More details are presented in section 4.

1.3.1 Extensions to the Sesame Web Interface

There were few minor extensions to the Sesame web interface necessary to allow the basic tracking functionality to be used. On the screenshot below, the new action "Set Version" which was added to the "Read Actions" and appears only for repositories configured with Tracking SAIL. When used, the form that can be seen in the lower frame of the screenshot appears.

The user can either label a new version either set as a working version one of those already labeled. To label a new version, one should select update ID out of those listed in the combo-box, provide a name for the new version in the "Label" text field, and press the Create button. After this action, the only effect is that there is a new state of the repository labeled as a version and it will appear in the list of versions. Next the user is redirected to the "home" form, the one seen immediately after selecting the repository.

If the user wants to the set as a working version, older version of the repository, s/he has to use the "Set Version" action and click on the link of the desired version. In this case, an auxiliary read-only branch of the repository is being created and automatically selected. The consequent read

operations will work with the desired old state of the repository. To switch back to the current/latest state of the repository, the user has to select the original repository again, using the "select other" link in the control frame. The auxiliary branch will be available until the user session that created it is "alive" (the sessions are being automatically discarded after a timeout-based strategy).

ontotext.com - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Logged in: **System account** [log out]
Repository: **OT, Enersearch Tracking** [select other]

Read actions: RQL RDQL Extract Explore Set version
Modify actions: Add (www) Add (copy-paste) Remove Clear

Select Version to work with or Label particular Update

About repository: **OT, Enersearch Tracking**.

Select Update ID to create a new Labeled Version

Label: ... a new version name ...

UpdateID: 6 Create

Repository Versions

		Select current version
1	System account	Auto Version after the first Upload
9	System account	update9

Internet

2 Integration and Remote Access

One of the important features that OMM adds to Sesame is the multi-protocol client access. At present the following options for integration of and remote access to OMM/Sesame are supported:

- HTTP requests – this option is the one originally supported by Sesame and lies underneath the standard web-interface to Sesame. Any applications capable in performing HTTP requests can use this option, according to the request specification presented in http://sesame.aidministrator.nl/publications/communication.html. Further, Java applications can make use of the client library wrapping the HTTP requests within Java classes and methods. Documentation of this library can be found at: http://sesame.aidministrator.nl/publications/api/client/. The HTTP access is further discussed in a sub-section below;
- Built-in use – Sesame can be directly built (or embedded) into Java applications. In this case there is no remote access protocol involved, both Sesame and the application using it work on one and the same machine within one and the same process. Obviously, this scenario for Sesame use is the most efficient one as far as there are no communication overheads. The Built-in usage of Sesame is addressed in a sub-section below;
- RMI calls – Sesame can also be accessed from Java applications via the RMI protocol. In contrast to HTTP, RMI is a well-developed RPC protocol. The communication overhead is much lower compared to HTTP or SOAP, but it is only feasible for Java applications. It is discussed in detail in a sub-section below;
- SOAP messages – Sesame can be used as a SOAP service from any platforms and programming languages. SOAP is an XML based RPC protocol that can work on top of HTTP. It has the advantages of both HTTP (being cross-platform) and RMI (being a true RPC protocol) combined with wide industry support. It has to be considered that the communication overhead for SOAP is even bigger than for HTTP. The SOAP access to Sesame is discussed in a sub-section below.

2.1 The OMM API, Reference and Public Server

The OMM API is a combination of three groups of methods:

- Standard Sesame interfaces – those which are available in Sesame via HTTP, namely: `addDataFromUrl(…)`, `clearRepository(…)`, `evalRdqlQuery(…)`, `evalRqlQuery(…)`, `extractRDF(…)`, `removeStatements(…)`, `selectRepository(…)`, and `uploadData(…)`;
- SAIL methods – methods from the SAIL API that provide lower level access to the repositories. Those are methods like `getSubClassesOf(…)`, `isProperty(…)`, etc. This methods provide more direct access to the repositories with respect to its basic RDF(S) semantic and graph representation without dependence on any query languages. See section 1.2.1 for better understanding of the SAIL role or the Sesame documentation for reference;
- OMM specific methods – such related to versioning and security sub-systems, meta-information, and other aspects of the knowledge control system (KCS). Such examples are `branchState(…)`, `revertToState(…)`, `getMetaInfo(…)`. A full list of these methods is presented in section 5 of [Kiryakov et al. 2002].

The OMM API is currently partially supported. There is support implemented for all the standard Sesame interfaces, part of the SAIL methods, and part of the OMM-specific methods. The standard interfaces are supported for all integration options. The Built-In use option provides easy access to all the SAIL methods, however, only part of those are available through RMI and SOAP, and non

of them via HTTP.

2.1.1 Correspondence with the Sesame's Java library for HTTP Clients

There are two major differences between the Java Client Library for HTTP (JCLH) access to Sesame developed by Aidministrator, and the OMM API:

- JCLH covers only the standard Sesame interfaces (the first group of methods, mentioned above);
- JCLH is available only in Java, and only for HTTP. In contrast, the OMM API is designed and implemented so to be efficient across different platforms and protocols.

Currently there are differences between the Client Library and the OMM API even where they overlap and can be aligned. The methods in OMM are equivalent to those in the Client Library, as far as both comply with the standard HTTP interface to Sesame. However, there are some differences in the parameters and result types and structures. Here the major differences are discussed:

- The OMM API is much more session-oriented than the JCLH and thus it counts more on the session context. As far as each session works on behalf of a single user with single repository in each moment, this information is kept in the session context rather than passed as parameters during each call. So, most of the JCLH methods take the repository ID as a parameter, while the OMM methods do not.
- The JCHL methods are implemented so to allow more information about the request execution, error messages, and transactions – information available through the `AdminListner` interface. In contrast, the OMM API is using fairly simple error codes, which make the communication faster and simpler, but provide less control and information.
- The JCHL is designed to allow streaming the results of query execution –feature extremely important for queries which execution takes a lot of time as well as in cases when the data processing is organized in a pipeline. In contrast, OMM is designed so to support in the most efficient way possible "normal" queries, returning reasonable amount of data within reasonable time. For instance, in JCLH, the RQL query execution can be performed via:

  ```
  QueryResultsTable evalRqlQuery(String query, String repository)
  ```

  while in the OMM API it comes like:

  ```
  String[][] evalRqlQuery(String query)
  ```

OntoText Lab, the developer of OMM, will keep working on better alignment of the two interfaces, which however will be subject of extensive consulting and co-ordination with aidministrator b.v., the developers of Sesame and the JCHL. Ideally, at the end, both JCHL and the OMM API should be as much compatible as possible, providing all the benefits of the different design approaches, so, to allow the application programmer to choose the proper method or structure for the specific task and application.

2.1.2 OMM API Reference

Here follows list of the methods currently supported for Built-in use, RMI and SOAP access. For some of the protocols, some of the methods may have slightly different parameters and return types, please, refer to the documentation for the specific protocol.

The methods are given in a JavaDoc-like notation in alphabetical order. Those, which are part of the standard Sesame interface are marked with STANDARD, those coming from the SAIL interface are marked with SAIL, those related to the OMM (actually the, Knowledge Control System), with KCS, and finally there are some auxiliary methods marked with AUX.

WARNNING! The KCS methods related to tracking of changes are only available for repositories

with tracking changes support. If, in addition, the repository is subject of access control, only the users with `History` permissions can successfully execute these methods. In all other cases the execution will fail or return no result.

Method Summary

`int`	`addDataFromUrl(String dataUrl, String baseURL)` Adds RDF(s) given URL of the RDF(s) document and a base URL. Returns the number of the statements added. **STANDARD**
`String`	`branchState(long stateUID)` Branch the repository at given state for further operations, returns the ID of the branched repository. **KCS**
`boolean`	`clearRepository()` Clears the repository. **STANDARD**
`void`	`continueCounterIncrement()` Continues with the normal increment of the update counter on each modification made in the repository. See `pauseCounterIncrement()`.**KCS**
`String[][]`	`evalRdqlQuery(String query)` Evaluates an RDQL query and returns the result. **STANDARD**
`String[][]`	`evalRqlQuery(String query)` Evaluates an RQL query and returns the result. **STANDARD**
`String`	`extractRDF(boolean ontology, boolean instances, boolean explicit)` Extracts the ontology and/or instances from the repository. **STANDARD**
`Vector`	`getClasses()` Retrieve the list of all the classes defined in the repository. **SAIL**
`Vector`	`getClassesOf(String anInstance, boolean mostSpecific)` Retrieve the list of classes to which `anInstance` belongs. According to the flag `moreSpecific` flag, the most specific ones can only be retrieved. **SAIL**
`Vector`	`getInstancesOf(String aClass, boolean proper)` Retrieve a list of URI's of an instances of a specific class. If the `proper` flag is set to `true`, the instances of its sub-classes are not retrieved. **SAIL**
`Map`	`getMetaInfo` (String subj, String pred, String obj) Retrieves the meta-info for a given statement. The statement is specified by the URIs of it's subject, predicate and object. The resulting map consists of keys mapped to the actual meta-info values. Generally, the keys are URIs from the KCS schema. E.g. meta-info key-value pair stating that a particular statement has vanished from the repository at a specific update (id): key : http://www.ontotext.com/otk/2002/03/kcs.rdfs#bornAt value : 3 **KCS**
`Vector`	`getProperties()`

	Retrieve the list of all the properties defined in the repository. **SAIL**
Vector	**getStatements**(String subj, String pred, String obj, boolean explicitOnly, boolean objIsLiteral) Retrieve a list of the statements from the repository, according to the pat. **SAIL**
Vector	**getSubClassesOf**(String resource, boolean direct) Retrieve the list of the sub-classes of a class. **SAIL**
Vector	**getSubPropertiesOf**(String resource, boolean direct) Retrieve the list of the sub-properties of a property. **SAIL**
Vector	**getSuperClassesOf**(String resource, boolean direct) Retrieve the list of the super-classes of a class. **SAIL**
Vector	**getSuperPropertiesOf**(String resource, boolean direct) Retrieve the list of the super-properties of a property. **SAIL**
long[]	**getUpdateIds**() Retrieve the list of the ids of all Updates of the repository, sorted in chronological order. The latest, i.e. the current, state is last. **KCS**
Map	**getUpdateMetaInfo** (String updateId) Retrieves the meta-info for a given update, specified by its id. The resulting map consists of keys (URIs from the KCS schema) mapped to the actual meta-info values. **KCS**
Vector	**getVersionIds** () Retrieves a Vector over the version ids of all the versions of the repository. **KCS**
Map	**getVersionMetaInfo** (String versionId) Retrieves the meta-info for a given version, specified by its id. The resulting map consists of keys (URIs from the KCS schema) mapped to the actual meta-info values. **KCS**
boolean	**hasStatement**(String subj, String pred, String obj, boolean explicitOnly, boolean objIsLiteral) Query the repository for particular statement. **SAIL**
boolean	**isClass**(String resource) Check that a URI is a class. **STANDARD**
boolean	**isInstanceOf**(String anInstance, String aClass, boolean proper) Check that a URI is an instance of a class. **SAIL**
boolean	**isPausedCounterIncrement**() Check if the update counter is paused. **KCS**
boolean	**isProperty**(String resource) Check that a URI is a property. **SAIL**
boolean	**isSubClassOf**(String subClass, String superClass, boolean direct) Query for subsumtion of two classes. **SAIL**

boolean	isSubPropertyOf(String subProperty, String superProperty, boolean direct) Query for subsumption of two properties. **SAIL**
void	labelCurrentState(String label) Create a labeled version of the current repository state. **KCS**
void	labelState(long stateUID, String label) Create a labeled version for a state of the repository assigning the necessary meta-information. **KCS**
String[]	listRepositories() Lists all the repositories. **KCS**
boolean	login(String userID, String pass) Logs in given a user and a password. **AUX**
void	pauseCounterIncrement() Stop the increment of the update counter, this way multiple updates will appear to be done "simultaneously". **KCS**
int	removeStatements(String subjURI, String predURI, String objURI, boolean bObjectIsLiteral) Removes statements corresponding to the `<subject, predicate, object>` pattern. Returns the number of deleted statements. The appropriate implicit statements are also removed, i.e. those inferred from and "supported" only by the statements being removed with this call. **STANDARD**
Void	revertToState(long stateUID) Restore the repository to previous state removing all statements added after the value of the update counter and revive all remover ones. **KCS**
boolean	selectRepository(String repository) Selects a repository to work with. **AUX**
int	uploadData(String data, String baseURL) Uploads RDF(s) in Sesame. Returns the number of statements added. **STANDARD**
void	workWithState(long stateUID) Switch the repository to a given previous state for the consequent read operations. The state is identified by update ID (UID) – like those returned from `getUpdateIDs()` method. This operation could be slow for big repositories, because an auxiliary repository (say, AUXR1), which is a *read-only* branch of the current one (say R1), is being created and automatically selected. To start working again with the current state, select the R1 repository via `selectRepository()`. **KCS**

2.1.3 OMM Public Server

OntoText Lab. supports a public OMM/Sesame server, which can be used for evaluation and research purposes. It can be accessed via the standard web-interface (or programmatically with HTTP requests) directly at http://omm.ontotext.com. This address is being currently redirected to http://62.213.161.156:8888/sesame - this address can change through the time, but on the hand its use will be faster and more reliable. SOAP and RMI access is also possible to this server. In case the server is out of service, new repository is needed, or other problems, feel free to contact

us at info@ontotext.com.

For test purposes one may use the "Sirma Skills KB" repository (with ID "mysql-sskb"), which is populated with the example presented in section 3 of [OntoText, 2002].

2.2 Built-in Usage of OMM and Sesame

This section explains how to build-in/embed the *OMM* and *Sesame* in another project or front-end Java application.

Since in this mode it is possible to access all the methods of the SAIL interfaces a complete JavaDoc reference is needed to use it. It is available at http://www.ontotext.com/OMM/StandAloneSesame/completeJavaDoc/index.html . All the classes that implement the **nl.aidministrator.rdf.sail.Sail** interface could be used in a stacked SAIL architecture defined in the configuration file used.

A class demonstrating this kind of usage (**builtin.BuiltInSesame**) is available at http://www.ontotext.com/OMM/StandAloneSesame/StandAloneSrc.zip. In the same archive there is an example configuration file (**system.conf**) which we suggest to be placed in the work folder of the project. In a set of experiments it demonstrates the usage of OMM/Sesame by performing a connection to our demo public server. The first example demonstrates usage of Sesame through the **RequestRouter** class; the second experiment uses directly the SAIL interface and the third one uses the **VersionManagement** interface to demonstrate the versioning and tracking changes capabilities added to Sesame by the OMM.

The class `com.ontotext.omm.util.SesameStartup` is used in the first experiment in order to initialize the stacked interfaces and to provide access via the **RequestRouter** class. **SesameStartup** provides a set of static initialization methods that should be used to initialize Sesame/OMM. The technical documentation of the stand-alone startup class is available at http://www.ontotext.com/OMM/StandAloneSesame/SesameStartupDoc/index.html.

The initialization via `com.ontotext.omm.util.SesameStartup` requires one of the following (i) a configuration file name, (ii) configuration file handler class and map of initialization parameters, (iii) a configuration file name and a parser class for it. It has been demonstrated with the first option.

2.2.1 Dependencies

This section specifies the dependencies of Sesame/OMM on third party libraries. In order to use Sesame\OMM in a built-in mode, the following libraries are required:

- `sesame.jar` – the classes of Sesame/OMM
- `jena.jar` – RDF(S)/DAML parser
- `rdf-api-XXX.jar` – RDF API
- `xerces.jar` (or XML parser)
- the JDBC drivers allowing access to the repository (in the example source this is mm.mysql-2.0.4-bin.jar).

All the third party libraries related to Sesame/OMM are available at http://www.ontotext.com/omm/downloads.html.

2.2.2 Initialization

A static invocation of one of the `initialize(…)` methods of the

`com.ontotext.omm.utils.SesameStartup`

class initializes the stacked SAILs. For instance:

```
SesameStartup.initialize("C:\\tomcat\\webapps\\sesame\\WEB-INF\\system.conf");
```

Afterwards the class `nl.aidministrator.rdf.config.SystemConfigCenter` (java doc reference available at http://www.ontotext.com/OMM/StandAloneSesame/SystemConfigCenterDoc/index.html) is used to gain access to the first stacked SAIL in a specified repository. This is demonstrated in the second example in the `builtin.BuiltInSesame` class available at http://www.ontotext.com/OMM/StandAloneSesame/StandAloneSrc.zip.

2.2.3 Access APIs

The access to Sesame/OMM is based on the

`nl.aidministrator.rdf.req_router.RequestRouter`

and

`nl.aidministrator.rdf.config.SystemConfigCenter` interfaces. From which one could retrieve any of the stacked SAILs.

2.3 RMI Access

RMI is supported as an alternative access method to Sesame. A couple of interfaces and their according server side implementations are used to provide an RMI access to Sesame, those reside in the `com.ontotext.omm.rmi` package. The entry point to Sesame is `FactoryInteface`. Its only method is used to gain access to the `ServicesInterface` through which the rest of the Sesame services are available. A standalone or servlet-based Sesame application can bind an instance of the `FactoryIntefaceImpl` only if an `rmiregistry` (an executable, part of the JDK distribution) is running on the server. Both interfaces should be compiled with the `rmcp` utility so the proper stub and skeleton code to be generated. Refer to the RMI documentation for full coverage of the interface compiling issue. Once the stub/skeleton code is present it could be packed with the factory and services interfaces in a *jar*. The jar should be added to the class path of the `rmiRegistry` application in order to bind/unbind, create, and use the Sesame RMI interfaces.

2.3.1 Client

This section contains a sample use case of Sesame via RMI. The first thing is to get a reference to the RMI Registry running on the server. This is performed via the call:

```
java.rmi.registry.Registry reg = LocateRegistry.getRegistry(RMI_HOST);
```

Where `RMI_HOST` is pointing to the server on which the RMI Registry is running. E.g. our demo server `dell.sirma.bg`

Afterwards, reference to the `FactoryInterface` should be retrieved from the newly located registry:

```
com.ontotext.omm.rmi.FactoryInterface f =
     (com.ontotext.omm.rmi.FactoryInterface)
     reg.lookup("FactoryInterface");
```

The next step is to get access to the `ServicesInteface` via the retrieved `FactoryInteface`:

```
com.ontotext.omm.rmi.ServicesInterface si = f.getService();
```

After this the services available could be accessed in the client application. A sample application, which could be seen as an example in the `testrmisesame.TestSesameRMI` class at http://www.ontotext.com/OMM/RMI/RMIClientSrc.zip.

The lifetime of a session is the same as the lifetime of the `ServicesInteface` instance returned from `getServices()`from the `FactoryInterface`. The JavaDoc documentation of the `ServicesInterface` is available at http://www.ontotext.com/OMM/RMI/ServicesInterface/index.html.

2.4 SOAP Access

OMM allows Sesame to be accessed through SOAP. The methods (or calls or messages) supported are analogous to those supported for the rest of the protocols, for price references check the Java client documentation. This section presents instructions for OMM/Sesame server installation as well as for SOAP access from client applications.

2.4.1 Server

In order to access Sesame via SOAP a fresh copy of the apache SOAP implementation should be retrieved from http://xml.apache.org/soap (or alternative implementation of the SOAP API). Afterwards the installation instructions should be followed in order to get SOAP running on a previously installed servlet server (e.g. Jakarta/Tomcat). We experienced problems to access the running Sesame servlet based application via SOAP interface with versions of Jakarta/Tomcat server less than 4.0, so, these instructions concern versions above 4.0.

Some changes to the installation of Sesame are required, in order to make it available via the SOAP interface. Instead of placing `sesame.jar` in the `$TOMCAT_HOME\webapps\sesame\WEB_INF\lib` folder[8], place it in the `$TOMCAT_HOME\common\lib` or in the folders mentioned in the `-Djava.endorsed.dirs` system property. This makes all the Sesame classes available for the SOAP running context. Once Sesame and SOAP have been installed and started, the Sesame SOAP service should be deployed on the server. Refer to the SOAP documentation for that issue and use the prepared DeploymentDescriptor.xml (http://www.OntoText.com/OMM/SOAP/DeploymentDescriptor.xml) to deploy the Sesame SOAP service.

2.4.2 Client

Client applications from different platforms written in all most popular programming languages can access OMM/Sesame through SOAP. Here we present SOAP access to OMM from Java application, including instructions, client library, and a sample application.

The following libraries are required:

- `soap.jar` – from the Apache SOAP site (http://xml.apache.org/soap);
- `mail.jar` – from http://java.sun.com/products/javamail/;
- `activation.jar` – from http://java.sun.com/products/beans/glasgow/jaf.html;
- An XML parser is required too.

In order for an application to be able to make SOAP calls to Sesame, the `com.ontotext.omm.soap.SoapClient` wrapper class should be instantiated, using the specific server URL and service ID. The client should login on the server and select a working repository.

The sources of the simple client library (the above mentioned `SoapClient` class) and a sample application (the `soaptest.Testsoap` class) that connects to a hard-coded server and repository are available at http://www.ontotext.com/OMM/SOAP/SoapClientSrc.zip. The documentation of the SOAP Client is available at http://www.ontotext.com/OMM/SOAP/SoapClientDoc.

---

[8] In more recent Jakarta versions, >=4.0.1, `TOMCAT_HOME` has been changed to `CATALINA_HOME`

2.5 Comaprative Protocol Performance Analysis

An important performance issue is the difference in the access speed through the various access methods (RMI, SOAP, HTTP, Built-In use). For each of these access methods, identical test cases have been performed, separated in the following three phases:

- *Initialization Phase* during which the respective access method is being initialized. This phase also includes login, password and database specifications, as well as access-method-specific setup information (e.g. URL of the server, configuration file, etc.)
- *Query Phase* in which two RQL queries are sequentially executed. The first one, being a simple *give-me-all-triples* query, and the second one a more complex *path* query.
- *SAIL API Phase* in which invocation of some of the SAIL methods takes place. This phase is not performed for the HTTP protocol, since the SAIL API is not available through it. The SAIL methods used in this phase are *getClasses()* and *getStatements(…)*.

The sources of this access methods performance measurement experiment are available at http://www.ontotext.com/omm/accessmetrix/accessmetrix.zip. The experiment could be easily started remotely since it uses the http://omm.ontototext.com demo server and has a configuration file (*system.conf*) included in it. The configuration file (*system.conf*) should be placed in the project's working folder. All the required third party libraries could be found at http://www.ontotext.com/omm/downloads.html.

In the following table, the results of the performance experiment are being presented in a normalized form. The metrics are not in any particular time unit, but instead use as a basis the access time for the respective Built-In phase, which is considered equal to 1 (one) virtual time unit. These results are formed on the average metrics after three identical executions of the experiment. The initialization phase is not presented because in all cases it is a matter of seconds, which is irrelevant for most of the applications.

Access Methods Performance Metrics

	Built-In	*HTTP*	*RMI*	*SOAP*
Query Phase	**1.00**	**1.04**	**1.09**	**1.78**
SAIL API Phase	**1.00**		**1.33**	**4.87**

Note: The times for Query and SAIL phases are not comparable to each other although both are presented as a same value for "Built-in" usage – the two sorts of usage are complementary rather than directly comparable.

It is easily seen that the fastest way to use Sesame/OMM is the Built-In use, which is intuitively expected. Under this integration scenario, the application is directly calling SAIL methods within the same process, i.e. without any sort of remote access.

As for using RQL queries HTTP and RMI perform in a similar manner. The reason is that the real query processing time at the server is a degree higher than the auxiliary time necessary for the calls needed to accomplish the query. SOAP is significantly slower in the Query Phase, which is also expected because of the XML serialization.

The Built-In use option performs even better in the direct SAIL API usage phase, where even RMI is way behind, not to mention SOAP. This can be explained with the fact that the SAIL methods are relatively fast, because they are much closer (than the queries) to the RDF data-model and the structure of the underlying database. In this case, the auxiliary time for "transportation" of the call, serialization or marshaling the attributes and so on is considerably big compared to the time for the

essential processing at the server.

3 Integration with OntoEdit

The OMM integration with OntoEdit had two purposes: to provide better integration between tools in the OTK toolsets and to test and prove the consistency of the OMM API. As a side effect, it also provided additional experience in building OntoEdit plug-ins.

OMM was integrated to OntoEdit by means of a plug-in module, taking benefit from the extensible architecture of the editor. The OMM plug-in was implemented on top of the existing Sesame plug-in with the following differences:

- The basic versioning functionality of OMM was made accessible;
- The connection between OntoEdit and OMM/Sesame was implemented through the RMI protocol. RMI was chosen as the most appropriate option because it is standard comprehensive remote procedure call (RPC) protocol on one hand, and it is specially designed for interoperation between Java applications. Thus it is faster than SOAP (see sub-section 2.5) and more convenient than HTTP.
- The RdfFileWriter and RdfFileReader plug-ins were used instead of the obfuscated RdfReader and RdfWriter classes used in the SesameClientPlugin.

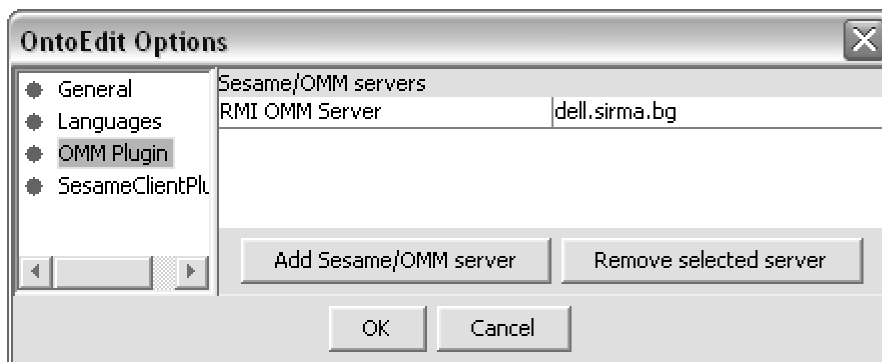
In a nutshell the Sesame Plug-in allows OntoEdit to load and store ontologies from Sesame rather than from files. The OMM plug-in in addition allows the versions and updates of the repository to be explored and one of those to be chosen for download. Those two plugins will be consolidated in the future so to provide maximum functionality and avoid duplication of efforts.

3.1 Availability

The OMM Plug-in jar along with the modified `ontoedit.bat`, and `omm.xml` (where our demo RMI server is configured by default) can be retrieved from http://www.ontotext.com/omm/OntoEditPlugin/OMMPlugin.zip. The sources of the OMM Plug-in are available at http://www.ontotext.com/omm/OntoEditPlugin/OMMPluginSrc.zip. All the required third party libraries and the last stable sesame.jar could be found at http://www.ontotext.com/omm/downloads.html. The free version of OntoEdit can be downloaded at http://www.ontoprise.de/com/start_downlo.htm. The plug-in can be installed following the standard procedure for plug-in installation in OntoEdit http://www.ontoprise.de/download/ontoedit_tutorial.zip.

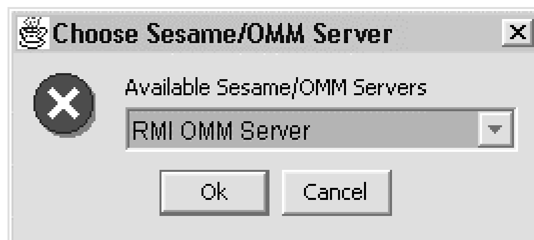
3.2 Quick User's Guide

Here follows a quick guide for the plug-in. In order to set the servers to be available for the OMM Plug-in the *View -> Options...* menu has to be chosen. By selecting the OMM Plug-in item one could manage the available servers for the plug-in.

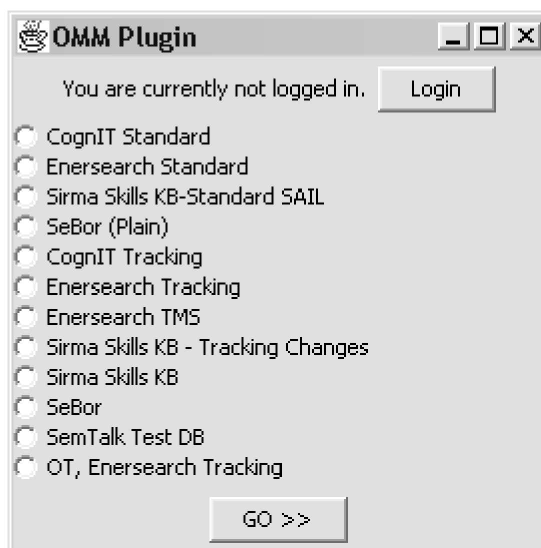


By pressing the *Add Sesame/OMM server* button name and URL of the server can be entered sequentially.

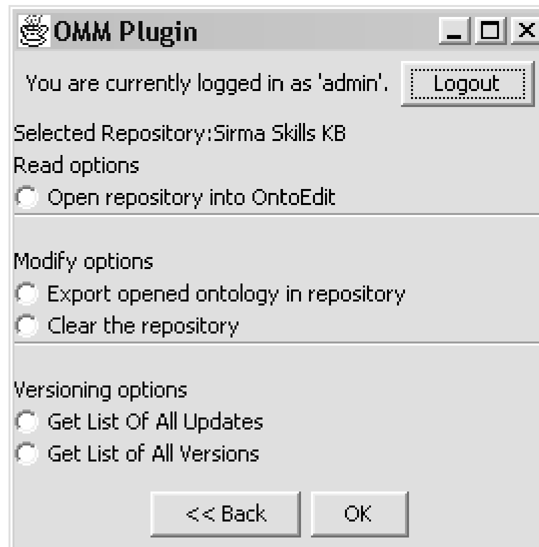
Once the plug-in has been set up, a button "Connect to Sesame/OMM" should have appeared on the OntoEdit's toolbar. The plug-in is started after pressing this button, and selection of a server to connect to is requested.



After selecting one of the available servers and pressing the OK button – a list of all repositories is being displayed.



At this point, by pressing *Login* one is able to provide user and password. E.g. [admin/admin] or

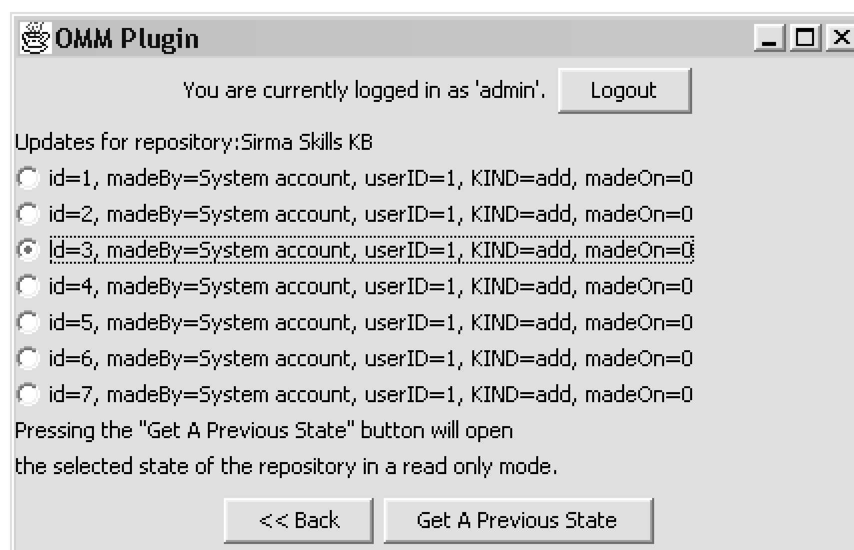


[anonymous/] for the OMM RMI demo server. After login is successful a repository should be chosen and the *Go >>* button pressed to view the available actions for this repository.

By selecting one of the Read, Modify or Versioning options and pressing the OK button various operations on the repository could be performed. The first two groups are identical to those with the Sesame plug-in. *Open repository into OntoEdit* extracts a repository from the remote server and opens it in OntoEdit. *Export opened ontology in repository* uploads a repository from OntoEdit to the remote server. *Clear the repository* deletes all the triples from the currently opened repository on the server.

The *Versioning options* provide the choices of getting a list of all updates or versions (labeled updates). Selecting an update of the repository to work with is accessible on the next step from the both versioning options available.

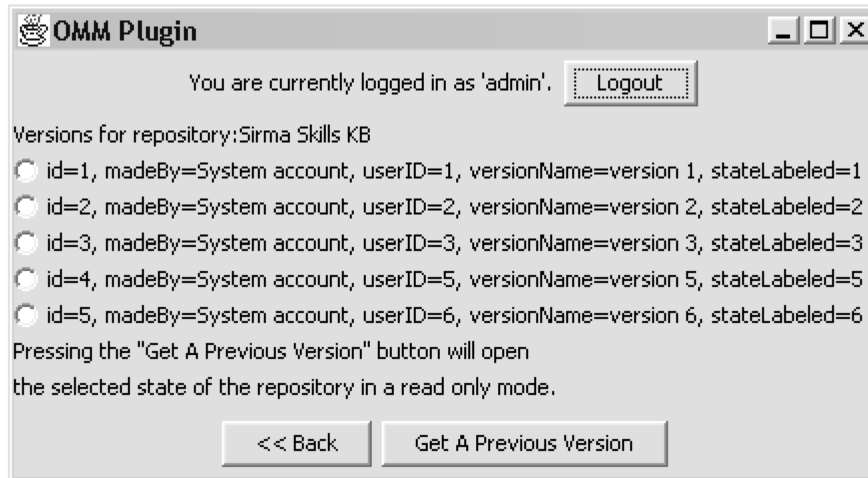
Choosing *Get List of All Updates* and pressing OK displays all the updates of the repository (generated, for instance, by sequential uploads).



The available meta-information for each update is being displayed on the same row after the

update id. By selecting an update id and pressing *Get A Previous State* a state of the repository after the update chosen is being loaded into OntoEdit in a read only mode. After this operation the OMM Plug-in is being closed to avoid confusion.

From the state of the plug-in allowing Read, Modify and Versioning options to choose from one could also select *Get List of All Versions* and press OK. This displays all the versions (labeled states) of the repository.



The available Meta-Info for each version is being appended to the version id on each row. By selecting a version id and pressing *Get A Previous Version* the selected version of the repository (technically speaking, after the update for which the version is a label) is being loaded into OntoEdit in a read only mode. After this operation the OMM Plug-in is automatically closed.

4 Integration of BOR reasoner with Sesame

BOR DAML+OIL reasoner was integrated with SESAME according to the schema outlined during the analysis and design phase (see [Kiryakov et al, 2002]) and provided for reference in section 1.2.2 here. BOR can be used through Sesame[9] in a way as simple as setting up new repository with the SeBOR SAIL included in the stack.

Our approach counts on the so called pre-reasoning – when new knowledge is uploaded into a BOR-ed[10] repository, BOR calculates the taxonomy[11] and classifies all the resources/instances with respect to the DAML+OIL semantics. The result is asserted as `subClassOf` and `type` statements in the repository, so, that SESAME can understand and use it for all further operations. More details about this integration are provided in this section.

4.1 Motivation

From a description logic's point of view RDF(S) has rather week notion of classes and means of expressing relationships between them. Namely RDF(S) utilizes simple named classes (no class expressions) and the only way to relate them is through the `subClassOf` property. And while the `subClassOf` relationship tends to be sufficient it is actually the lack of class expressions that makes the ontology development in an RDF(S) environment such a hard task to accomplish.

That was the main purpose of BOR's integration into SESAME – ease the process of ontology development and make the resulting taxonomy and classification available to SESAME users at all levels.

4.2 Design Overview

BOR uses the same repository for both explicit 'definition' statements provided by the user(s) and 'qualification' or 'pre-reasoned' statements inferred by it. For instance, if it can be inferred from the ontology that `C` is direct sub-class of `D`, than `<C, subClassOf, D>` statement will be added in the same repository where the 'original' statements of the ontology reside. Although this mixture is useful from Sesame point of view it raises the requirement for efficient means of distinguishing statements of those two disjoint groups.

For that purpose a new set of properties is introduced with the schema

http://www.ontotext.com/otk/2002/07/bor

Properties defined in this schema are said to be BOR-specific or special properties. Statements utilizing a special property are said to be special statements. These special predicates are exclusively used in pre-reasoned statements. BOR treats every non-special explicit statement in the repository as a definition.

When BOR builds the taxonomy the immediate super classes of a class are stated using the `bor:subClassOf` property. Having this property is defined to be a sub-property of `rdfs:subClassOf`, according to the model-theoretic semantics, the RDFS inference engine adds the appropriate implicit statements, such as `<A, rfds:subClassOf, B>`, where `<A,`

[9] It is question of perspective, in can also be thought that Sesame takes benefit from BOR's understanding of DAML+OIL.

[10] We use "BOR-ed repository" as a shortage from "BOR-enabled repository", i.e. a Sesame repository in which SAIL stack the SEBOR SAIL was included.

[11] The minimal sufficient set of direct `subClassOf` relations, which allows all valid subsumptions to be derived because of the transitivity of the relation.

`bor:subClassOf, B>` statement is available plus the transitive closure of the `rdfs:subClassOf` statements. The described approach has the following advantages:

- Clear separation between definitions and pre-reasoned qualifications;
- Easy extraction of immediate sub/super-classes of a given class;
- Easy to clear qualification subpart of the repository;
- The structure of the taxonomy is explicitly available to other SESAME layers

Analogously, when the result from the instance classification is stored in the repository the special property `bor:type` (a sub-property of `rdf:type`) is used to state the most specific types of an instance.

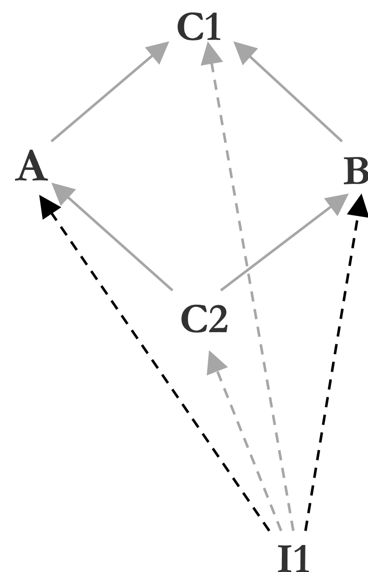
4.3 Usage Scenario

A typical usage of BOR would be:

1. Upload ontology and instance data through the standard OMM/Sesame interfaces;
2. BOR will be automatically invoked to build the taxonomy and the classification and store them into the repository as already explained;
3. Then repeatedly access the repository in any read-only manner (e.g. RQL queries)

This scenario can be demonstrated as follows. Imagine we have a DAML+OIL repository (ontology plus some instance data) as presented below:

- Primitive classes: **A**, **B**
- Classes defined as follows: **C1=A ∪ B**, **C2=A ∩ B**
- **I1** is asserted as an instance of both the primitive classes, i.e. **I1:A** and **I1:B**



After the uploading of the above knowledge, BOR infers that:

- **C1** is super class of both **A** and **B**
- **C2** is a sub-class of both **A** and **B**
- **I1** is instance of **C1** and **C2**, i.e. **I:C1** and **I:C2**

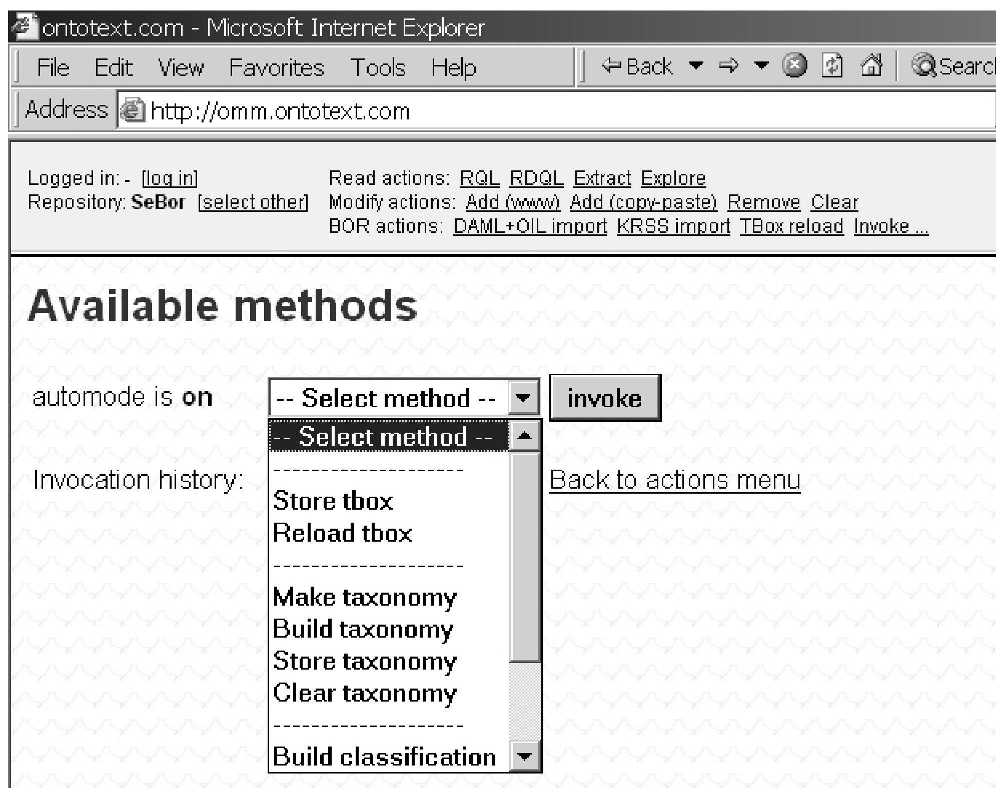
At the diagram, the instantiation links are given with dashed lines, while the subsumption links with non-dashed. The black links are the one explicitly defined, while the orange those inferred by BOR.

This approach will be inefficient for applications for which the update operations are relatively more often compared to the read operations. However, the integration of BOR with Sesame provides the most easy to use DAML+OIL-aware platform compared to the “competitors”:

- The ontology editor OILEd (http://oiled.man.ac.uk/) coupled with FACT reasoner (http://www.cs.man.ac.uk/~horrocks/FaCT/). There is no query language within this setup – the only action possible is to check the consistency of an ontology. FACT itself has no DAML+OIL import;
- TRIPLE (http://triple.semanticweb.org/) – an RDF query, inference, and transformation language for the Semantic Web. It supports DAML+OIL in combination with RACER and FACT reasoner. Again there is no support for intensive DAML+OIL modification scenario.
- OpenCyc, "Taxonomic inferences provided by OpenCyc for DAML ontologies" available at http://opencyc.sourceforge.net/daml/daml-taxonomic-inferences.html. A similar approach is taken, however, OpenCyc was still not available when publishing this document.

5 User interface and demo

As a typical back-end tool BOR has no much UI. When a BOR-ed repository is used through the Sesame web-interface, the upper pane get extended with third line of actions ("BOR actions", as seen at the screenshot below) which provide access to the full power of the reasoner.



Two BOR-ed repositories "Sebor" and "Sebor Plain" are available for experiments at the OntoText public OMM Server at http://omm.ontotext.com

5.1 Known Issues

As expected, number of issues appeared to be problematic with respect to DAML+OIL and its schema. Some of those (as the first one below) appeared to be general problems rather than just an implementation issue for BOR.

5.1.1 *Inconsistency: Literal sameClassAs Literal in DAML+OIL schema*

It is a problem with the so-called "Importing terms from RDF/RDFS" part of the DAML+OIL schema, which appears even without the MTS (Model-Theoretic Semantics) of RDFS. The "Importing terms" as they are now are wrong and it seems nobody imported nothing automatically using them.

Here is the problem:

1. It is defined in the schema that

```
<daml:Class, subClassOf, rdfs:Class>
```

if we simplify it a bit, it is because the RDF(S) notion for class is much weaker – it includes any sorts of classes in contrast to the DAML+OIL one which takes just "object classes", without

datatype classes (such as `Literal`) and relation classes (such as `Property`).

2. What is said in the "Importing terms"

```
<rdfs:Class rdf:ID="Literal">
  <sameClassAs rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
</rdfs:Class>
```

is intuitively OK, and can distilled into triples as follows:

```
<daml:Literal, type, rdfs:Class>

<daml:Literal, sameClassAs, rdfs:Literal>
```

3. However the domain and range of `daml:sameClassOf` according to the schema are

```
<sameClassOf, domain, #Class>

<sameClassOf, range, #Class>
```

and as part of the DAML+OIL schema `#Class` means `daml:Class`

4. From 3 both the MT semantics of RDF(S) and (all sorts of) DAML+OIL semantics support the inference that:

```
<X, sameClassAs, Y> => <X, type, daml:Class> and <Y, type, daml:Class>
```

If 3 and 4 correctly represent the intended semantics of `sameClassAs`, i.e. sth that declares two daml+oil "object classes" being equivalent, than no one should use it for `rdfs:Class`-es, without realizing that it implicitly defines them being also `daml:Class`-es. This is the case with Literal, as follows

5. From 2 and 4 it follows that:

```
<rdfs:Literal, type, daml:Class>
```

which is confusing because it is much stronger (and incorrect) than what we already know about the `rdfs:Literal`, namely

```
<rdfs:Literal, type, rdfs:Class>
```

It is the very same problem with the way `rdfs:Property` is imported in the DAML+OIL schema.

What can be done?

- First option: the definition of `sameClassOf` should be relaxed wrt to the domain and range restrictions, becoming:
  ```
  <sameClassOf, domain, rdfs:Class>
  <sameClassOf, range, rdfs:Class>
  ```
  It seems to us, however, that this is going to be problematic with the MTS of DAML+OIL.
- Second option: If `sameClassOf` domain and range restrictions cannot be relaxed, than the Importing Terms in the DAML+OIL specification have to be reformulated in a way, especially those for `Literal` and `Property` classes. It can be handled as follows:
  ```
  <daml:Literal, type, rdfs:Class>
  <daml:Literal, equivalentTo, rdfs:Literal>
  ```
  And even better we can more precisely say that `Literal` is a data type class rather then a general one, so, replacing the first triple above with:
  ```
  <daml:Literal, type, daml:Datatype>
  ```
  So, finally, we propose the import of `Literal` to look like:
  ```
  <Datatype rdf:ID="Literal">
  ```

```
    <equivalentTo rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
</Datatype>
```

For the sake of completeness, it can also be added that

```
<daml:Literal, subClassOf, rdfs:Literal>
```

5.1.2 Why we need separate properties for definitions and qualifications?

Let us consider the following ontology:

```
(def-c Parent-of-happy-child (some has-child Happy))
(def-c Parent-of-rich-child (some has-child Rich))
(def-pc Proud-parent
(or Parent-of-happy-child Parent-of-rich-child))
```

Its RDF(S) triple representation will be:

```
("Happy"                 rdf:type            daml:Class)
("Rich"                  rdf:type            daml:Class)

("Parent-of-happy-child" rdf:type            daml:Restriction)
("Parent-of-happy-child" daml:onProperty     "has-child")
("Parent-of-happy-child" daml:hasClass       "Happy")

("Parent-of-rich-child"  rdf:type            daml:Restriction)
("Parent-of-rich-child"  daml:onProperty     "has-child")
("Parent-of-rich-child"  daml:hasClass  "Rich")

("Proud-parent"          rdf:type            daml:Class)
("Proud-parent"          rdfs:subClassOf     "_0")

("_0"                    rdf:type            daml:Class)
("_0"                    daml:intersectionOf "_1")

("_1"                    rdf:type            daml:List)
("_1"                    daml:first          "Parent-of-happy-child")
("_1"                    daml:rest           "_2")

("_2"                    rdf:type            daml:List)
("_2"                    daml:first          "Parent-of-rich-child")
("_2"                    daml:rest           daml:nil)
```

Where `_0`, `_1`, etc. are IDs reserved for anonymous resources. Note that as a result of the addition of these statements the repository will infer and add a number of (implicit) statements.

Now, let us suppose at this point BOR is invoked to build the taxonomy. As this is rather expensive operation the result (i.e. the taxonomy tree) will be stored in the repository. Eventually this will add the following RDF(S) statements to the repository:

```
(daml:Nothing            rdfs:subClassOf    "Proud-parent")
("Proud-parent"          rdfs:subClassOf    "Parent-of-happy-child")
("Proud-parent"          rdfs:subClassOf    "Parent-of-rich-child")
("Parent-of-happy-child" rdfs:subClassOf    daml:Thing)
("Parent-of-rich-child"  rdfs:subClassOf    daml:Thing)
```

Note that the repository will infer the transitive closure of the `subClassOf` property. The resulting state of the repository will not be eligible for loading into BOR again. Take, for example, `Proud-parent`. Repository will contain at least the following statements about it:

```
("Proud-parent"          rdfs:subClassOf     "Parent-of-happy-child")
("Proud-parent"          rdfs:subClassOf     "Parent-of-rich-child")
```

```
("Proud-parent"         rdfs:subClassOf     "_0")
```

There are certain problems:

Problem 1 How to determine which of the statements are definitive and which are qualificative (i.e. pre-reasoned by BOR)?

Problem 2 The extraction of the immediate sub/super classes of a given class will be much too cumbersome due to the transitive closure of the `subClassOf` property the repository has introduced. (Note: this is not quite right – a distinction on the base of the 'explicit' flag is possible)

The proposed solution

Introduce a number of custom properties that Bor should utilize in order to make a distinction between definitions/qualifications:

```
(definedAsSameClassAs   rdfs:subPropertyOf  daml:sameClassAs)
(definedAsSubClassOf    rdfs:subPropertyOf  daml:subClassOf)

(immediateSubClassOf    rdfs:subPropertyOf  daml:subClassOf)
(equivalentClassTo      rdfs:subPropertyOf  daml:sameClassAs)
```

5.2 Improvement Plans

Currently any modification to a BOR-ed repository causes the whole taxonomy and classification to be recalculated which is quite ineffective. It would be very useful to inspect the type of modification done and if it does not concern the terminological part of the repository – do not rebuild the taxonomy and do not reclassify all but only the affected individuals.

6 Experience with the Enersearch Case Study

The Enersearch case study within On-To-Knowledge was designed so to involve most of the tools within the toolset for an organizational memory application of a virtual enterprise[12]. In essence, the documents are processed with OntoExtract and OntoWrapper so to automatically generate RDF descriptions that were uploaded into Sesame. On top of this, Spectacle was used for information access based on the RDF descriptions. The huge volume of the descriptions provides a good test case for the load-tolerance of the involved tools. For our experiments we used a sample dataset from the case study with about 83000 RDF(S) statements.

6.1 Alignment of the RDF(S) File Structuring

Having a body of RDF and RDFS knowledge stored in files provides a lot of flexibility for its organization. Additional freedom comes from the different options for organization of the namespaces that can be shared between files as well as used in parallel within a single file.

Mismatches in the namespace usage policy may however cause some problems. For instance, the "Updative" mode for RDFS upload supported by OMM, allows newer version of a file previously uploaded in the same repository, to intelligently update the repository, rather than to be added as an independent body of knowledge. The implementation of this updative mode requires distinct base URLs to be used for the different files during the upload. This problem together with number of tricky RDFS parsing issues were detected and resolved during the experiments involving the Enersearch data and OMM.

[12] To enable the distributed team of Enersearch to be able to easily share and lookup documents.

6.2 Performance Observations

Here follow some statistics on the upload time for the RDF files generated by OntoExtract in the course of the Enersearch case study.

		Standard		TMS (only)		Tracking	
	State-ments	total time (s)	stm/ sec	total time (s)	stm/ sec	total time (s)	stm/ sec
home.rdf	236	2,7	87	22	11	22	11
EMMSEC00-1.rdf	5 132	19	270	480	11	510	10
contacts.rdf	328	3,3	99	27	12	31	11
ises1_utility_utility.rdf	6 457	25	258	737	9	750	9
ises2.rdf	4 824	15	322	699	7	680	7
ises6.rdf	4 143	16	259	581	7	616	7
ises5_impact_impact-m3.rdf	3 931	16	246	540	7	560	7
ises5_rtlc_benefits6-m3.rdf	3 942	15	263	520	8	561	7
ises7.rdf	4 356	16	272	780	6	772	6
ises7_EU98_eu98-t_html.rdf	9 751	33	295	1920	5	1920	5

The results from uploading the first few files (with the number of RDF statements in the second column) into three differently configured repositories at one and the same server:

- **Standard** – it is a Sesame repository running on top of MySQL without truth maintenance system (TMS);
- **TMS (only)** – it is again a Sesame repository with TMS working, which means in this configuration allows consistent delete operations. The TMS is not necessary in case of static repositories;
- **Tracking** – a repository configured with the OMM SAIL for tracking of changes.

The statistics for small files are not always representative because of problems with precise measurement of the time needed. As it is easily seen, the TMS is a fairly expensive feature with its current implementation. This experiment confirms the observations from [Iosif and Mika, 2002] that the speed for upload in repositories with TMS depends in a logarithmic dependency from the number of the statements already uploaded in the repository. The tracking of changes takes insignificant time compared to TMS.

6.2.1 In-Memory SAIL

Further experiments were conducted within the same installation and knowledge base, but using the so-called in-memory SAIL (the graph-based one) currently in development from Aidministrator. The idea of this SAIL is to boost the performance as much as possible based on a sort of total-caching strategy where disk files are used only to assure persistency of the knowledge, but not for querying. The current implementation has no TMS and Tracking.

Even taking the fact that this reasoning option is still under development the results are indicative for the final result. The average speed achieved was 1023 statements/sec, i.e. more than three times faster than the Standard option above. In the same time, the in-memory representation of all the data required about 20MB or average of 250 bytes/statement. This demonstrates that even for knowledge bases and ontologies ten or more times bigger than the current example, the in-memory SAIL will be a very feasible alternative of the standard once based on relational databases.

6.3 Lessons Learned

There were number of lessons learned experimenting with different setups of the repository for the Enersearch case study:

- A text indexing application which uses RDF(S) repository for storage and retrieval of the appropriate meta-information is characterized with huge volumes of statements even for small documents. This confirmed our expectations based on engineering experience in related areas such as corpora linguistics and information extraction;
- Even reasoning and storage platform based on ontology language as simple as RDF(S), when coupled with truth maintenance system shows scalability problems. This problem becomes obvious for implementations based on straightforward access to relational database for the essential data operations.
- In-memory implementation of the core RDFS operations is feasible for volumes much bigger than those that can be handled efficiently with database-bound processing.
- Applications with similar volumes of data using ontology language substantially more expressive than RDFS (such as DAML+OIL and other description logics) are inefficient for real-time usage.

7 Conclusion

During the last phase of the project the Ontology Middleware Module (OMM) was further developed, polished, and integrated within the On-To-Knowledge tool set. The experiments within the framework of the Enersearch case-study helped a lot for tuning and debugging of the implementation of a number of the KCS features. The integration with OntoEdit served well for identifying and correcting some inconsistencies in the OMM API which would be hard to discover otherwise. In general, the easy integration of OMM within the tool set proven once again the vitality of the overall concept for building set of tools interoperating on the basis of standard knowledge representation and methodology (in contrast to the tightly integrated suites or complex systems).

It is more than obvious that further usage of OMM in more applications and contexts will be crucial for its maturity, however, the technological risk for those is already acceptable after the integration with the rest of the tools, the lessons learned and the appropriate corrections in the technology.

8 References

[Brickley and Guha, 2000] W3C; Dan Brickley, R.V. Guha, eds.
Resource Description Framework (RDF) Schemas.
http://www.w3.org/TR/2000/CR-rdf-schema-20000327/

[Broekstra and Kampman, 2001a] Jeen Broekstra, Arjohn Kampman
Query Language Definition.
Deliverable 10, On-To-Knowledge project, May 2001.
http://www.ontoknowledge.org/downl/del10.pdf

[Broekstra and Kampman, 2001b] Jeen Broekstra, Arjohn Kampman
Sesame: A generic Architecture for Storing and Querying RDF and RDF Schema.
Deliverable 9, On-To-Knowledge project, October 2001.
http://www.ontoknowledge.org/downl/del10.pdf

[Cycorp, 1997] Cycorp. Cyc Public Ontology, 1997.
http://www.cyc.com/cyc-2-1/

[Ding et al, 2001] Ying Ding, Dieter Fensel, Michel Klein, Borys Omelayenko
Ontology management: survey, requirements and directions.
Deliverable 4, On-To-Knowledge project, June 2001.
http://www.ontoknowledge.org/downl/del4.pdf

[Iosif and Mika, 2002] Victor Iosif and Peter Mika (EnerSearch AB)
On-To-Knowledge: EnerSearch Virtual Organization. Case Study: Evaluation Document.
Deliverable 29 On-To-Knowledge project, September 2001.
http://www.ontoknowledge.org/downl/del29.pdf

[Kiryakov and Ognyanov, 2002a] Atanas Kiryakov and Damyan Ognyanov.
Tracking Changes in RDF(S) Repositories.
In the Proceedings of Workshop on Knowledge Transformation for the Semantic Web, at the 15th European Conference on Artificial Intelligence, pages 27-25. Lyon, France, July 23, 2002.

[Kiryakov and Ognyanov, 2002b] Atanas Kiryakov and Damyan Ognyanov.
Tracking Changes in RDF(S) Repositories.
In the Proceedings of 13th International Conference on Knowledge Engineering and Knowledge Management EKAW02, Sigüenza, Spain, 1-4 October 2002. To Appear.

[Kiryakov et al, 2002] Kiryakov A., Simov K. Iv., Ognyanov D. *Ontology Middleware: Analysis and Design.* Deliverable 38, On-To-Knowledge project, March 2002.
http://www.ontoknowledge.org/downl/del38.pdf

[OntoText, 2002] OntoText Lab; Kiryakov A., Popov B., Ognyanov D., *Ontology Middleware: System Documentation.* July 2002.
http://www.ontotext.com/omm/OMMDocumentation.pdf

[Lassila and Swick, 1999] W3C; Ora Lassila, Ralph R. Swick, eds.
Resource Description Framework (RDF) Model and Syntax Specification
http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/

[Simov and Jordanov, 2002] Simov K. Iv., Jordanov S. *BOR: a Pragmatic DAML+OIL Reasoner.* Deliverable 40, On-To-Knowledge project, June 2002.
http://www.ontoknowledge.org/downl/del40.pdf

[Novotny and Lau, 2001] Novotny B., Lau T. *Organizational MemoryDescription of Case Study Prototypes.* Deliverable 20, On-To-Knowledge project, June 2002.
http://www.ontoknowledge.org/downl/del20.pdf