

Ontology Middleware: Analysis and Design

Atanas Kiryakov, Kiril Simov, Damyan Ognyanov

OntoText Lab.

Identifier	38
Class	Deliverable
Version	1.0
Version date	March 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 Searchlsruhe, Germany; Schweizerische Lebensversicherungs- und Rentenanstalt / Swiss Life, Switzerland; British Telecommunications plc, UK; CognIT a.s, Norway; EnerSearch AB, Sweden; Administrator 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 Searchlsruhe, Germany
Tel: +49-721-608392
Fax: +49-721-693717
Contactperson: R. Studer
E-mail: studer@aifb.uni-searchlsruhe.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

Administrator 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@administrator.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

This report outlines the requirements for the Ontology Middleware Module (to be developed in task 11.2) and the reasoning enhancements (to be developed in task 11.3), as well, as the appropriate conceptual models and implementation approaches proposed as a result of the research, analysis, and discussions that took place within the consortium.

Architectural design of the new modules is presented so to specify the roles of the modules and subsystems and the interfaces to be supported. The expressivity of the language and the services to be supported by the reasoner are also specified.

Our analysis builds on top of the research already conducted in the course of the project, and more precisely deliverables numbers 1-4, 6, 9, 10, and 15-17 of the On-To-Knowledge project.

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), Administrator Nederland BV (NL), OntoText Lab. (BG). We wish to particularly thank to the Arjohn Kampman, Frank van Harmelen, Jeen Broekstra, Marin Dimitrov, and Michel Klein for the instant support and the fruitful discussions.

Contents

1	Introduction	6
1.1	Ontology Middleware	7
1.2	Ontologies vs. Knowledge Bases.....	7
1.3	Scope.....	8
1.3.1	Managing Repositories, RDF Terminology	8
1.4	Ontology Representations in RDFS.....	8
2	Tracking Changes, Versioning, and Meta-Information.....	11
2.1	Related Work	11
2.2	Requirements	11
2.3	Versioning Model for RDF(S) Repositories	12
2.3.1	History and Versions	14
2.4	Meta-Information.....	15
2.4.1	Model and Representation of the Meta-Information	15
2.4.2	Tracking Changes in the Meta-Information	15
2.5	Implementation Approach	16
2.5.1	Batch Updates.....	17
2.5.2	Versioning and Meta-information for Imported Statements	17
2.5.3	Versioning and Meta-information for Inferred Statements	17
2.5.4	Versioning of Knowledge Represented in Files	18
2.5.5	Branching Repositories	19
2.5.6	Controlling the History.....	19
2.6	Formal Representation of the Meta-Information	20
2.6.1	Meta-Information for Statements	21
2.6.2	Low-level Representation of Meta-Information for Statements.....	22
3	Security and Access Control.....	24
3.1	Requirements	24
3.2	Related Work	24
3.3	Fine-grained Access Rights Support Business Logics Definition	26
3.4	Basic Principles of the Security Model.....	27
3.4.1	Security Classes.....	29
3.4.2	Query Restrictions	29
3.4.3	Three Layers of Complexity and Support	30
3.5	Implementation Approach	31
3.5.1	Standard Security Restrictions	31
3.5.2	Security Classes Support	32
3.5.3	Query Restriction Type Support.....	33
3.5.4	Formal Representation of the Security Information	33
4	Instance Reasoning in On-To-Knowledge.....	35
4.1	Reasoning in On-To-Knowledge Project.....	35
4.1.1	Discussion on Architecture with Separate Storage and Reasoning	35
4.1.2	Requirements Towards a Reasoning Service for On-To-Knowledge	36
4.1.3	DAML+OIL Reasoner Built-In the Repository.....	36

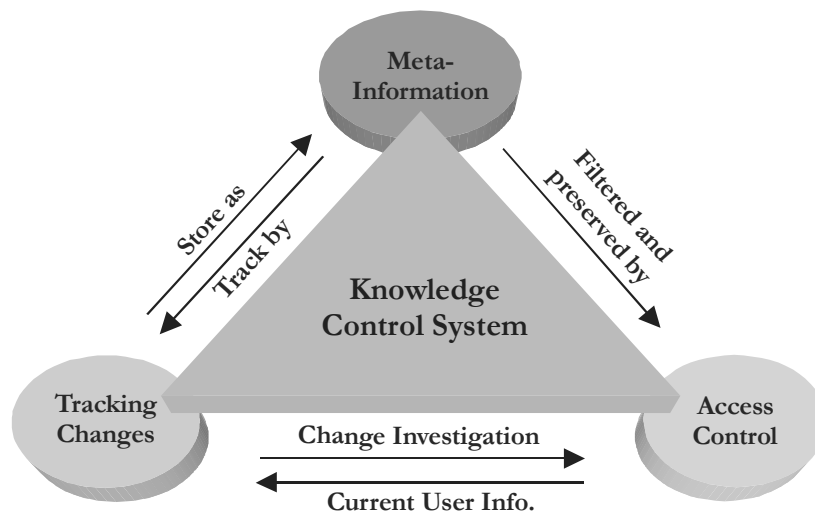
4.2	DAML+OIL and RDF(S)	36
4.2.1	The Incompatible Semantics	37
4.2.2	Incompatible Reasoning	38
4.2.3	Separation of DAML+OIL from RDF(S).....	38
4.3	DAML+OIL as a Description Logic.....	39
4.4	Inference Services in Description Logics	40
4.4.1	Terminological Reasoning.....	40
4.4.2	Instance Reasoning	40
4.4.3	Deterministic versus non-deterministic	41
4.4.4	Extending versus non-extending	41
4.5	Instance Reasoning in On-To-Knowledge Project	41
4.5.1	Realisation	42
4.5.2	Instance checking	42
4.5.3	Retrieval	42
4.5.4	Retrieval of components.....	43
4.5.5	Model Checking	43
4.5.6	Minimal Sub-Ontology Extraction.....	43
4.6	Implementation	43
4.6.1	The structure of the ontology and instance representation	44
4.6.2	The control over the inference process.....	44
4.6.3	Datatypes	44
4.7	Functional Interfaces to a DAML+OIL Reasoner	44
4.7.1	Tell Interfaces.....	45
4.7.2	Delete Interfaces.....	45
4.7.3	Ask Interfaces.....	46
4.7.4	Display Interfaces.....	47
5	Architecture and Interfaces.....	48
5.1	The Initial Design and Why it Changed	48
5.2	Overview of the Current SESAME Architecture.....	48
5.3	How OMM Fits in the Picture?.....	49
5.4	Meta-Information.....	50
5.5	Versioning Components, Version Management, and Tracking Changes	51
5.5.1	VersionManagement interface.....	51
5.5.2	Version interface	51
5.5.3	TrackingChanges interface.....	52
5.6	Security Components. Architecture	52
5.6.1	KCS Interface	53
5.6.2	SecurityServices Interface	53
5.6.3	User Interface	54
5.6.4	Role Interface	54
5.6.5	Rules.....	54
5.6.6	Restriction Interfaces.....	54
5.6.7	SecurityClass Interface.....	56
5.7	Reasoner interface.....	56
6	References	57

1 Introduction

The middleware modules discussed in sections 2 and 3 of this document can be seen as „administrative“ software infrastructure that will make 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 bases¹. In the light of these objectives the following main features are targeted:

- 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. The dependencies are explained in the corresponding sections.



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. A KCS should be carefully designed so to support these two distinct use cases.

In addition to the „administrative“ modules, BOR (the reasoning module defined in section 4) extends the reasoning services of SESAME with features and interfaces that were recognized to be important for many applications such as knowledge exchange and integration and information extraction. The SAIL interface of the SESAME architecture is used for “plugging” of reasoner for language other than RDF(S), namely DAML+OIL. A specific functional interface (and the corresponding API) for description logic reasoners is specified so to provide access to the full power of such system in the most adequate and efficient way. Of course, the most important development is BOR, the DAML+OIL reasoner, itself.

¹ See the next sub-section for discussion on ontology vs. instance data vs. knowledge base.

1.1 Ontology Middleware

An ontology middleware system should serve as a flexible and extendable platform for knowledge management solutions. It has to provide infrastructure with 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 sub-section 5.2 as there is already some level of support for it in the SESAME architecture.
- Support for pluggable 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) so providing easy integration, interoperability between applications, knowledge maintenance and reuse.

The design of the ontology middleware module proposed here is just an extension of the SESAME architecture (see [Broekstra and Kampman, 2001b]) as described in sub-sections 5.2 and 5.3.

1.2 Ontologies vs. Knowledge Bases

A number of justifications in the terminology are necessary in order to ensure less confusion and better understanding of this document. An almost trivial but very important question is “What do the OTK tools support: ontologies, data, knowledge, or knowledge bases?” The simple answer is “All of this”. Here we will provide an improvised interpretation of these terms (otherwise discussed in depth in a huge number of sources).

Ontology is the basic knowledge formally defining the model (schema, conceptualization) relevant for certain situation. Here “situation” should be understood in its wide philosophic sense – any kind of mental context for which specific knowledge is applicable, including topics, subjects, scientific fields, professions, cultural societies. In practice, the ontologies usually represent models of knowledge domains which allow the information relevant to a more or less general set of applications considering this domain to be represented in as much as possible suitable, structured, and non-redundant way. The ontologies are usually defined in a kind of formal logical or schema-definition language and at least define the types of entities to be considered together with some characteristics, constraints, and relations applicable to them. An ontology may or may not include: type hierarchies, attribute definitions and restrictions, induction or other rules. Examples of ontologies could be a database schema or a definition of a product catalogue structure and categories. The ontology of a mail-server application could be its conceptual model that may include entities like other servers, messages, people (considered just as recipients), mailing lists, as well as information about their characteristics and possible relations between them. Only the explicitly and formally defined conceptual models are considered as ontologies – so, most of the mail-servers are not ontology based just because the model of the world they work with is not formally specified.

Data or **instance data** is the particular information to be managed – it usually considers specific situation or phenomena. In the case of the mail-server application, this could be any information related to a particular e-mail message, user, or mailing list. The data should always be structured according to the ontology (if such is defined, of course) and complies to its restrictions. In the ideal case a big number of different data sets and applications use one and the same ontology. While the instance data and the applications may regularly change so to reflect changes in the situation or particular needs, the ontologies are expected to be much more stable so to provide a coherent interpretation of the data and easy integration among the applications.

Knowledge is a hard to define philosophic category that is often interpreted in different ways

according to the context and purpose of use. It is the case that both ontologies and data can be considered as knowledge, so, it is often used as a generic term denoting both types of information. **Knowledge Base** is a term used in a way much analogous to Database, so, non-surprisingly it may denote both a specific body of knowledge as well as the software system for its management. In contrast to the databases, the knowledge base management systems are expected to support some inference mechanisms so to be able to make explicit facts that logically follow from the existing knowledge but were not formally asserted. For instance, if the fact "A is father of B" is asserted, a database should not be expected to return A as an answer of the question "Who are the parents of B?" while a knowledge base should be able to. However, it is important to be realized that this will be possible only in case the rule "if (X is father of Y) then (X is parent of Y)" is in one or another way defined in the system. Such assertions are the typical constituents of an ontology.

It is also the case that **Repository** is pretty often used as a synonym of Database or Knowledge Base or just as a more generic term that denotes any of them, as well, as hybrid and marginal systems. So, everything commented in the previous paragraph is also applicable for repositories.

1.3 Scope

The On-To-Knowledge project is concerned with research and development on a formal knowledge management methodologies, formalisms, tools, and case studies. The methodologies developed concern the full lifecycle of a knowledge-based system, starting from the domain analysis and ontology design and following all the steps including maintenance and revisions of both ontology and instance data. Many of the tools developed are also suitable for both ontologies and instance data, for instance, the OntoEdit editor and the Spectacle viewer.

The ontology middleware module should extend the SESAME RDF(S) repository that will affect the management of both ontologies and instance data in a pretty much unified fashion as far as those are not strictly separated in RDF(S). It means that versioning, access control, and meta-information will be supported for both kinds of knowledge.

The DAML+OIL reasoner (to be developed under the reasoning enhancements task) will provide support for both ontologies and instance data represented in DAML+OIL language which is an extension of RDF(S). The result will be more effective knowledge representation and deeper inference.

1.3.1 Managing Repositories, RDF Terminology

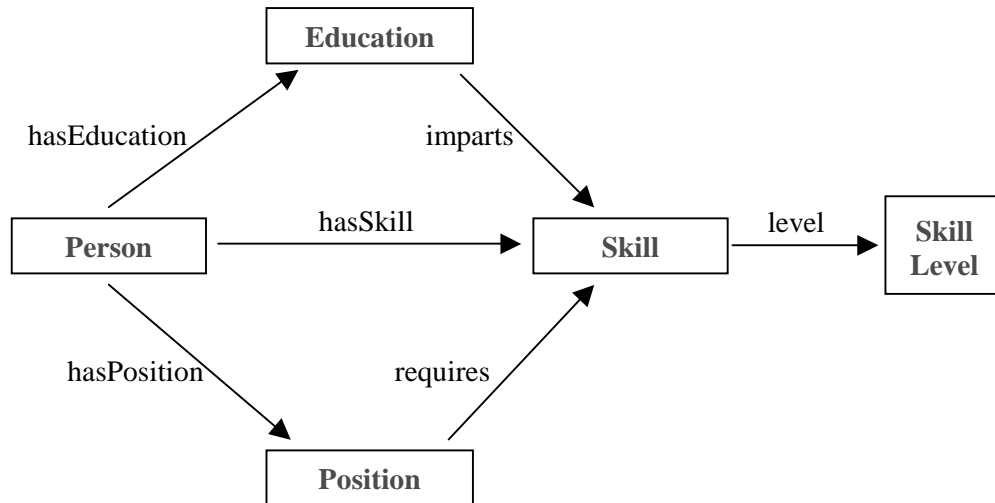
For the purpose of compliance with the terminology used in the SESAME RDF(S) repository, in this document the term **repository** will be used to denote a compact body of knowledge that could be used, manipulated, and referred as a whole. Such may contain (or host) both ontological assertions and instance data.

In the knowledge management community there is a rich diversity of almost equivalent terms used to reference pretty similar modeling primitives. Non-surprisingly, this is the result of the existence of a variety of different paradigms with their own vocabularies. Again for compliance in this document we will use mostly the RDF(S) terminology and when necessary DAML+OIL one. So, we will use: *Class* for any concepts, types and other unary predicates; *Property* for any binary relations, attributes, slots, etc. More detailed discussion on this issue could be found in [Kiryakov et al., 2001].

1.4 Ontology Representations in RDFS

As far as RDF(S) is widely used in the project and also provides the basis of the DAML+OIL specification, it is extremely important to at least shortly discuss its various forms and representations. RDF(S) is defined in [Lassila and Swick, 1999] and [Brickley and Guha, 2000]. Its semantic is most clearly defined in [Hayes, 2001].

Below we are going to present what a relatively simple ontology may look like from different perspectives. The sample we use here is based on the skill management case study presented in Deliverable 20. Let us consider such system managing the following classes of entities: *Person*, *Skill*, *Education*, *Position*, and *Skill Level*. A number of relations are also defined as follows: *hasSkill* (between Person and Skill), *hasEducation* (between Person and Education, the person has taken the appropriate course), *hasPosition* (between Person and Position), *requires* (between Position and Skill); *imparts* (from Education to Skill), and finally *level* (between Skill and SkillLevel). It has to be clarified that those are types of relations that can actually take place between instances of these classes. Some person PER1 may have a skill SKI1, which on its turn has level SKL3, an instance of a Skill Level or its class (say, Level 1, Level 2, etc.) This is presented in an ER-like fashion below.



Part of the same ontology (just the Person class and hasSkill property) represented in XML serialization of RDFS may look like:

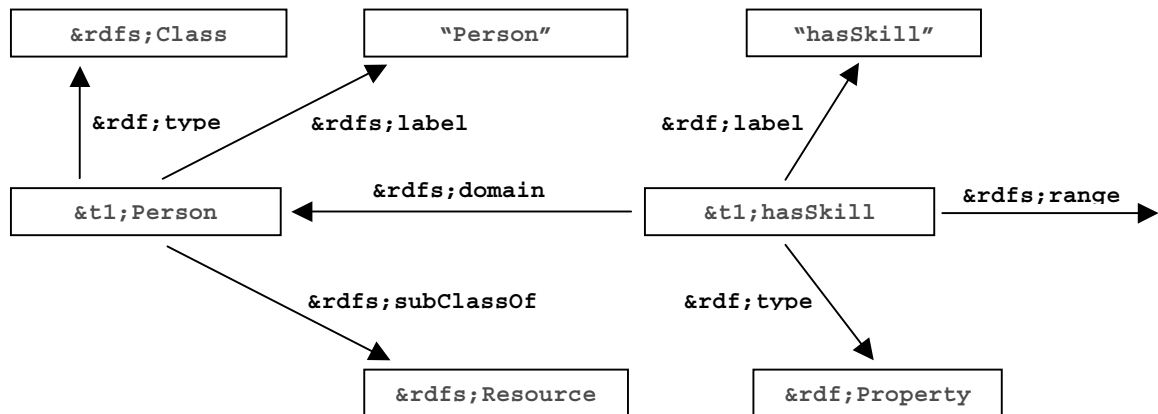
```

...
<rdfs:Class rdf:about="&t1;Person" rdfs:label="Person">
  <rdfs:comment>A Human Being ... </rdfs:comment>
  <rdfs:subClassOf rdf:resource="&rdfs;Resource"/>
</rdfs:Class>

<rdf:Property rdf:about="&t1;hasSkill" rdfs:label="hasSkill">
  <rdfs:domain rdf:resource="&t1;Person"/>
  <rdfs:range rdf:resource="&t1;Skill"/>
</rdf:Property>
...

```

The above XML representation should not let us forget that the basic (and in a way the only one) expressive mean of RDF is the Statement – that could be best represented as the following triple **<Subject, Predicate, Object>** where the first two elements are URIs of resources and the third one is either again an URI either a literal. The formal representation of a part of the ontology as an RDF graph may look like follows:



Each arc in the directed graph above corresponds to a single RDF statement (i.e. triple) where the source node is the Subject resource, the target node is the Object, and the arcs are labeled with the predicate resource. The two namespaces used above are mapped respectively: `&rdfs;` – to the RDFS definition; `&t1;` – to the definition of the specific ontology we discuss.

The only purpose of the above explanations on RDF(S) were to recall the various aspects of the knowledge representation there. More analytical study on the various XML-based ontology representation languages (including RDF(S) and OIL) could be found in [Dimitrov2000].

2 Tracking Changes, Versioning, and Meta-Information

The Big Brother is watching you!

1984, George Orwell

The problem for tracking changes within a knowledge base is addressed in this section. It is important to clarify that higher-level evaluation or classification of the updates (considering, for instance, different sorts of compatibility between two states or between a new ontology and old instance data) is beyond the scope of this work. Those are studied and discussed in depth in [Ding et al, 2001], sub-section 2.2. The tracking of the changes in the knowledge (as discussed here) provides the necessary basis for further analysis. As an example, in order to judge the compatibility between two states of an ontology a system should be able to at least retrieve the two states and/or the differences between them.

An overview of the related work followed by formal statement of the requirements is presented first. Next, a model that satisfies these requirements is presented complemented with comments on a possible implementation approach. In summary, the approach taken can be shortly characterized as "versioning of RDF on a structural level in the spirit of the software source control systems".

2.1 Related Work

Here we will shortly review similar work, namely, several other studies related to the management of different versions of a complex objects. In general, although some of the sources discuss closely related problems and solutions, there is not one addressing ontology evolution and version management in a fashion allowing granularity down to the level of statements (or similar constructs) and also being able to capture interactive changes in knowledge repositories such as asserting or retracting statements.

One of the studies that provide a methodological framework pretty close to the one need here is [Kitcharoensakkul and Wuwongse, 2001]. The authors model a framework, which is designed to handle the identification, control, recording, and tracking of the evolution of software products, objects, structures, and their interrelationships. The paper investigates the different models and versioning strategies for large scale software projects and present a way to express the meta-information and the impact of a single change over the various components of the project in RDF(S) – in this case used just for representation of the related meta-information, the object being followed are from the software domain.

Database schema evolution and the tasks related to keeping schema and data consistent to each other can be recognized as very similar to ours. A detailed and pretty formal study on this problem can be found in [Franconi et al, 2000a] – it presents an approach allowing the different sorts of modifications of the schema to be expressed within suitable description logic. More detailed information about the reasoning and other related tasks could be found in [Franconi et al, 2000b].

Another study dealing with the design of a framework handling database schema versioning is presented in [Benatallah and Tari, 1998]. It is similar to [Franconi et al, 2000a and 2000b] papers and can be seen as a different approach of handling the changes of the evolving object and the process of class evolution.

Unsurprisingly, some of the most relevant studies were done under the On-To-Knowledge project – among the reports concerning various aspects of the knowledge management, most relevant is [Ding et al, 2001], mentioned earlier in this section.

2.2 Requirements

The top-level requirements towards a versioning model for knowledge management in the contexts

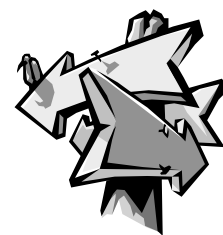
targeted by the On-To-Knowledge project are enumerated below:

- (i) To allow tracking of changes in both ontologies and instance data with fine granularity.
- (ii) To allow work with, revert to, extraction, and branching of different states and versions of a knowledge base. This to be possible without any loss or corruption of information related to other states or versions.
- (iii) To be independent from the serialization of the knowledge. In other words a change of the ordering or formatting of the knowledge that does not cause changes in its formal interpretation not to be considered as a change of the state. This requirement should not be understood in the sense of a necessity for continuous evaluations of logical equivalence, but rather as a need for abstraction from reordering of statements or terms and other similar cosmetic changes which are syntactically detectable as sense-preserving in a declarative representation.
- (iv) To be applicable for various platforms or medias. The mechanism to allow versioning of repositories (where statements can be asserted and retracted in an interactive way), but to be also adaptable for use cases when the different states of a repository (a body of knowledge) are represented as, say, XML files.
- (v) To be applicable for knowledge represented in RDF(S), DAML+OIL, and similar ontology languages.
- (vi) To be as much as possible independent from the semantics of a specific state or variant of a language. Taking into account the current state of the development of the Semantic Web vision and the related languages, specifications, and tools, development of a scheme particularly tuned for language such as DAML+OIL seems not to be feasible.
- (vii) To be simple and intuitive, optionally transparent, so, to allow use of the knowledge in a manner abstracting from the change tracking. Thus not to unnecessarily increase the complexity of simple applications.
- (viii) To allow versioning of the meta-information as well. Motivation provided in sub-sections 2.4.1 and 2.4.2.
- (ix) To allow naming, addressing, and keeping meta-information for specific states. Well-structured meta-information considering various characteristics of a certain state (or version) of a knowledge base to be supported in a flexible way.
- (x) To cause minimal overhead to the volume of the data. In principle, tracking the changes with fine granularity may lead to tracking information volume much bigger than the volume of the knowledge being tracked.
- (xi) To provide basis for further evaluation of the changes, such as:
 - Consistency of a state of the repository;
 - Various sorts of compatibility between states of ontology and data (like those discussed in [Ding et al, 2001].)

2.3 Versioning Model for RDF(S) Repositories

A versioning model for RDF(S) repositories is proposed. To make it more explicit (i) the knowledge representation paradigm supported is RDF(S) and (ii) what is being tracked are repositories – independently from the fact if they contain ontologies, instance data, or both. Few of the most important reasons that influenced this pretty basic decision follow:

- A kind of knowledge representation paradigm should be selected so to make the task well-defined and allow implementation. Such have to be backed by a



formally specified standard. The most widely accepted and general purpose candidates seem to be KIF, OKBC, and RDF(S);

- RDF(S) is a widely-accepted W3C standard. There are number of tools being able to import, process, manage, and export knowledge represented in RDF(S). This is not the case with OKBC because it is not formally recognized as a standard. It also does not hold for KIF, because it is an interchange format suitable for transferring knowledge in heterogeneous environments, but not for knowledge management within an application;
- DAML+OIL is defined as a RDF(S) extension, so, even though the semantic differs, the basic representation primitives are the same: classes, properties, resources. On the other hand DAML+OIL itself is still not standardized and widely supported. Also, as commented in Deliverable 2, there are concrete plans (in W3C and other organizations) for development of a new language, currently known as OWL, that will most probably succeed DAML+OIL;
- All the tools and technology currently developed in On-To-Knowledge are based on RDF(S). An exception to this rule is OntoEdit. What is most important, SESAME, the storage and reasoning facility developed in the project and used in all the case studies is an RDF(S) repository;
- As far as the ontologies (schemata) and the instance data are represented in a unified way in RDF(S), what should be supported is versioning for repositories that may contain any mixture of the both kinds of knowledge. It is an application-specific question how the various kinds of knowledge have to be structured and distributed between various repositories depending on methodological and practical considerations. We decided not to constrain this flexibility with assumptions for specific organization.

The decision to support tracking of changes, versioning, and meta-information for RDF(S) repositories has a number of consequences and requires more decisions to be taken. The most important principles are presented in the next paragraphs.

VPRI: *The RDF statement is the smallest directly manageable piece of knowledge.*

Each repository, formally speaking, is a set of RDF statements (i.e. triples) – these are the smallest separately manageable pieces of knowledge. There exist arguments that the resources and the literals are the smallest entities – it is true in a way, however they cannot be manipulated independently. It is the case that none of them can independently “live” in a repository because they always appear as a part of a triple and never independently. The moment when a resource was added to the repository may only be defined indirectly as the same as “the moment when the first triple including the resource was added”. Analogously a resource may be considered removed from a repository when the last statement containing it gets out. To summarize, there is no way to add, remove, or update (the description of) a resource without also changing some statements while the opposite does not hold. So, the resources and the literals from a representational and structural point of view are dependent from the statements.

VPR2: *An RDF statement cannot be changed – it can only be added and removed.*

As far as the statements are nothing more than triples, changing one of the constituents, just converts it into another triple. It is because there is nothing else but the constituents to determine the identity of the triple, which is an abstract entity being fully defined by them. Let us take for instance the statement $ST1 = \langle A, PR1, B \rangle$ and suppose B is a resource, i.e. an URI of resource. Then ST1 is nothing more but a triple of the URIs of A, PR1, and B – if one of those get changed it will be already pointing to a different resource that may or may not have something in common with the first one. For example, if the URI of A was `http://x.y.z/o1#A` and it get changed to `http://x.y.z/o1#C` then the statement $ST2 = \langle C, PR1, B \rangle$ will be a completely different statement.

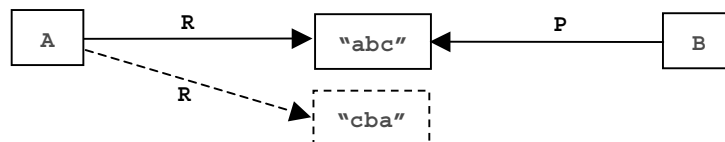
Further, if the resource pointed by an URI gets changed two cases could be distinguished:

- The resource is changed but its meta-description in RDF is not. Such changes are

outside the scope of the problem for tracking changes in formally represented knowledge, and particularly in RDF(S) repositories.

- The description of the resource is changed – it can happen iff a statement including this resource get changed, i.e. added or removed. In such case, there is another statement affected, but the one that just bears the URI of the same resource does not.

There could be an argument, that when the object of a triple is a literal and it gets changed, this is still the same triple. However, if there is for instance statement $\langle A, R, "abc" \rangle$ and it changes to $\langle A, R, "cba" \rangle$, the graph representation shows that it is just a different arc because the new literal is a new node and there could be other statements (say, $\langle B, P, "abc" \rangle$) still connected to the old one.



As a consequence here comes the next principle:

VPR3: *The two basic types of updates in a repository are addition and removal of a statement*

In other words, those are the events that necessarily have to be tracked by a tracking system. It is obvious that more event types such as replacement or simultaneous addition of a number of statements may also be considered as relevant for an RDF(S) repository change tracking system. However, those can all be seen as composite events that can be modeled via sequences of additions and removals. As far as there is no doubt that the solution proposed should allow for tracking of composite events (say, via post-processing of the sequence of the simple ones), we are not going to enumerate or specify them here.

VPR4: *Each update turns the repository into a new state*

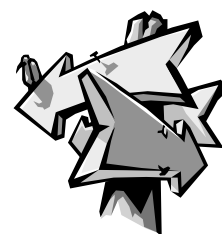
Formally, a state of the repository is determined by the set of statements that are explicitly asserted. As far as each update is changing the set of statements, it is also turning the repository into another state. A tracking system should be able to address and manage all the states of a repository.

2.3.1 History and Versions

Some notes and definitions that complement the above stated principles are presented below.

History, Passing through Equivalent States

The history of changes in the repository could be defined as sequence of states, as well, as a sequence of updates, because there is always an update that turned repository from one state to the next one. It has to be mentioned that in the history, there could be a number of equivalent states. It is just a question of perspective do we consider those as one and the same state or as equivalent ones. Both perspectives bear some advantages for some applications. We accepted that there could be equivalent states in the history of a repository, but they are still managed as distinct entities. Although it is hard to provide formal justification for this decision the following arguments can be presented:



- For most of the applications it is not typical a repository to pass through equivalent states often. Although possible, accounting for this phenomenon does not obviously worth taking into account that finding (or matching) equivalent states could be a computationally very heavy task.

- It is the case that if necessary, equivalent states could be identified and matched or combined via post-processing of a history of a repository.

Versions are labeled states of the repository

Some of the states of the repository could be pointed out as versions. Such could be any state, without any formal criteria and requirements – it completely depends on the user's or application's needs and desires. Once defined to be a version, the state becomes a first class entity for which additional knowledge could be supported as a meta-information (in the fashion described below.)

2.4 Meta-Information

Meta-information should be supported for the following classes: resources, literals, statements, and versions. As far as DAML+OIL ontologies are also formally encoded as resources (of type `daml:Ontology`) meta-information can be attached to them as well.

2.4.1 Model and Representation of the Meta-Information

We propose the meta-information to be modeled in RDF as well – something completely possible taking into account the unrestricted meta-modeling approach behind RDF(S) (see sub-section 4.2). A number of objections against such approach can be given:

- It increases the number of meta-layers and so it makes the representation more abstract and hard to understand. However, adding meta-information always requires one more layer in the representation, so, making it via extension of the same primitives used for the “real data” (instead of defining some new formalization) can even be considered as a simplification.
- It makes possible confusion and may introduce technical difficulties, say, because of intensive use of heavy expressive means such as reification.

The schema proposed below handles in some degree these problems and provides number advantages:

- It is probably the most typical role of RDF to be used for encoding of meta-information
- One and the same technology can be used for viewing, editing, and management of both knowledge and meta-information. Any RDF(S) reasoners and editors will be able to handle meta-information without special support for it.
- Queries including both knowledge and meta-information will be pretty straightforward. So, lookup of knowledge according to conditions involving both meta-information and “real” knowledge will be possible. Imagine a situation, when a complex ontology is being developed and there is meta-information supporting this process, say, a meta-property “Status” (with possible values “New”, “Verified against the ontology”, “Verified against the sample data”, “Done”) being attached to each class. Then a lookup of all classes that are subclasses of C and have status “New” will be just a typical query against the RDF repository.
- Representing the meta-information as RDF could be done in a flexible way that allows it to be customized for the specific needs of the use case.

Having analysed the above pros and cons we propose the schema outlined in subsection 2.5.

2.4.2 Tracking Changes in the Meta-Information

An important decision to be taken is whether changes in the meta-information should be tracked. The resolution proposed here is: Changes in the meta-information should be considered as regular changes of the repository, so, they turn it from one state to another. Here are few arguments backing this position:

- There are number of cases when the only result of a serious work over an ontology is just a single change in the meta-information. Let us use again the example with the “Status” meta-property for classes (described above.) The result of a complex analysis of the coherence of a class definition may result just in changing the status from “New” to one of the other values. In such case, although there is no formal change in the “real” data, something important get changed. From an ontology management point of view it is extremely important tracking of such changes to be possible.
- If necessary, it is possible appropriate analysis to be made so that changes that affect only meta-information to be ignored. This way both behaviors can be achieved. In case of the opposite decision (not to track changes in meta-information), no kind of analysis can reconstruct the missing information.
- An analogy with the software source control systems may also provide additional intuition about this issue. If we consider the comments in the software code as a meta-information, it becomes clear that the source control systems definitely consider changes in the meta-information as equal to the “real” changes in the code.

2.5 Implementation Approach

Let us first propose a schema for tracking changes in a repository. For each repository, there will be an *update counter* (UC) – an integer variable that increases its value each time when the repository is updated, that in the basic case means when a statement get added to or deleted from the repository. Let us call each separate value of the UC *update identifier*, UID. Then for each statement in the repository the UIDs when it was added and removed will be known – these values determines the “lifetime” of the statement. It is also the case that each state of the repository is identified by the corresponding UID.

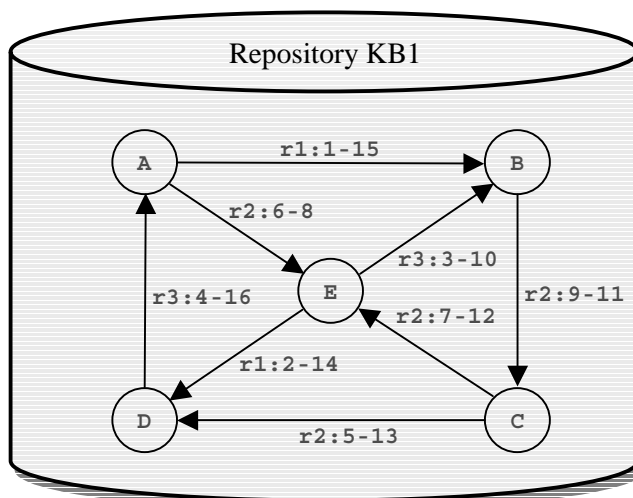
If the UIDs that determine the “lifetime” of each statement are kept, for each state it will be straightforward to find the set of statements that determine it – those that were “alive” at the UID of the state being examined. As far as versions are nothing more than labeled states, for each one there will be also UID that uniquely determines the version.

The approach could be demonstrated with the sample repository KB1 and its “history”. The repository is represented as a graph where the lifetime of the statements is given separated with semicolons after the property names. The history is presented as a sequence of events in format

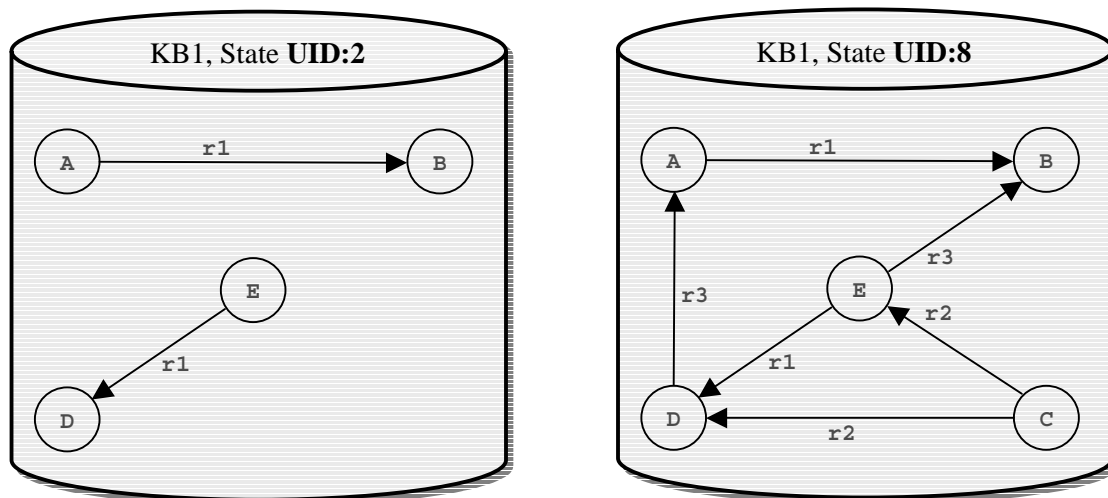
```
UID:nn {add|remove} <subj, pred, obj>
```

History:

```
UID:1 add <A, r1, B>
UID:2 add <E, r1, D>
UID:3 add <E, r3, B>
UID:4 add <D, r3, A>
UID:5 add <C, r2, D>
UID:6 add <A, r2, E>
UID:7 add <C, r2, E>
UID:8 remove <A, r2, E>
UID:9 add <B, r2, C>
UID:10 remove <E, r3, B>
UID:11 remove <B, r2, C>
UID:12 remove <C, r2, E>
UID:13 remove <C, r2, D>
UID:14 remove <E, r1, D>
UID:15 remove <A, r1, B>
UID:16 remove <D, r3, A>
```



Here follow two “snapshots” of states of the repository respectively for UIDs 2 and 8



It is an interesting question how we handle in the above model, multiple additions and removals of one and the same statement, which in a sense periodically appears and disappears from the repository. We undertake the approach to consider them as separate statements, because of reasons similar to those presented for the support of distinguishable equivalent statements.

2.5.1 Batch Updates

We call batch update the possibility the update counter of the repository to be stopped, so not to increment its value for a number of consequent updates. This feature can be very important for cases when it does not make sense the individual updates to be tracked one by one. Such example could be assertion of a DAML+OIL element that is represented via set of RDF statements none of which can be interpreted separately (see subsection 4.2.3).

Another example for a reasonable batch update would be an application that works with the repository in a transactional fashion – series of updates are bundled together, because according to the logic of the application they are closely related. Finally, batch updates can also be used for file imports (see subsection 2.5.4).

2.5.2 Versioning and Meta-information for Imported Statements

New statements can appear in the repository when an external ontology is imported in the repository either by `xmlns:prefix="uri"` attribute of an XML tag in the serialized form of the ontology either by `daml:imports` statement found in the header of a DAML+OIL ontology. In each of those cases the imported statements in the repository should be treated as read-only and thus the users of the repository cannot change them. All these statements will be added and removed to/from the repository simultaneously (with the same value of the `omm:bornAt` and `omm:diedAt` properties) as the statement that causes their inference or import. An additional note about the imported statements related to the security: these statements should be recognized as external, and not belonging to the repository and thus we can avoid the application of the security policies to them. Meta-information may not be attached to such statements.

2.5.3 Versioning and Meta-information for Inferred Statements

There are cases when addition of a single statement in the repository leads to the appearance of several more statements in it. For example, the addition of the statement `ST1=<B, rdfs:subClassOf, C>` leads to the addition of two new statements `ST2=<B, rdf:type, rdfs:Class>` and `ST3=<C, rdf:type, rdfs:Class>`. This is a kind of simple inference necessary

to “uncover” knowledge that is implicit but important for the consistency of the repository. There are number of such inferences implemented in SESAME.

The question about the lifetime of such inferred statements is far not trivial. Obviously, they get born when inferred. In the simplest case, they should die (get removed) together with the statement that caused them to be inferred. However, imagine that after the addition of ST_1 in the repository, there was another statement added, namely $ST_4 = \langle B, \text{rdfs:subClassOf}, D \rangle$. As far, as ST_2 is already in the repository only $ST_5 = \langle D, \text{rdf:type}, \text{rdfs:class} \rangle$ will be inferred and added. Now, imagine ST_1 is deleted next while ST_4 remains untouched. Should we delete ST_2 ? It was added together with ST_1 on one hand, but on the other it also follows from ST_4 . One approach for resolving such problems is the so-called “truth maintenance systems” (TMS) – basically, for each statement (or at least for the inferred ones) information is being kept about the statements or expressions that “support” it, i.e. such that (directly) lead to its inference.

There are two important things to mention. First, the implementation of TMS is problem of SESAME itself – it is not raised by the knowledge control system. Second, the implementation of a comprehensive TMS even for language such as RDF(S) is a very challenging task. There is already some work done in this direction in SESAME.

Suppose, there is a TMS working in SESAME, the tracking of the inferred statements is relatively easy. When the TMS “decides” that an inferred statement is not supported anymore, it will be deleted – this is the natural end of its lifetime. It will be considered as deleted during the last update in the repository, which automatically becomes a sort of batch update (if it is not already.)

As with the imported statements, meta-information may not be attached to inferred statements.

The security restrictions towards inferred statements can be summarized as follows:

- Inferred statements may not be directly removed;
- A user can read an inferred statement iff s/he can read one of the statements that support it.
- The rights for adding statements are irrelevant – a user may or may not be allowed to add a statement independently from the fact is it already inferred or not.

2.5.4 Versioning of Knowledge Represented in Files

The issues concerning the work with knowledge represented in files and its versioning can be discussed in two main topics – each of them presenting a different involvement of the content of the files.

The first one is the case when the SESAME uses a specific storage and inference layer (SAIL) to access knowledge directly from the underlying file – so files used for persistency instead of, say, relational database. In such case we cannot control the appearance and disappearance of distinct statements, which easily can happen independently from SESAME. The knowledge control system (KCS) presented here is not applicable for such SAILs.

The second case is to import knowledge into the SESAME from files – one of the typical usage scenarios. The first step is to convert the file F into a set of statements FS , which also includes the inferred ones. Next, the appropriate changes are made in the repository within a single batch update (see subsection 2.5.1.) Three different modes for populating repository from files are supported:

- **Re-initializing** – the existing content of the repository is cleared and the set of statement FS is added. No kind of tracking or meta-information is preserved for the statements that were in the repository before the update. This is equivalent to Clear followed by Accumulative import;
- **Accumulative** – FS is added to the repository, it actually means that the statements from FS that are already in the repository are ignored (any tracking and meta-information for

them remains unchanged) and the rest of the statements are added. This type of import has to be used carefully, because it may lead to inconsistency of the repository even if its previous state was consistent and the file was consistent;

- **Updating** – after the import the repository contains only the statements from the file, the set \mathcal{FS} (as in the re-initializing mode). The difference is that the statements from the repository that were not in \mathcal{FS} are deleted but not cleared, i.e. after the update, they are still kept together with the corresponding tracking and meta-information. The statements from \mathcal{FS} that were not already in the repository² are added.

The Updating import mode is the most comprehensive one and allows the repository to be used to track changes in a file that is being edited externally and “checked-in” periodically. This can also be used for outlining differences between versions or different ontologies represented in files.

2.5.5 Branching Repositories

Branching of states of repositories will be possible. In this case a new repository will be created and populated with a certain state of the existing one – the state we want to make branch of. When a state is getting branched, it will automatically be labeled as a version first. The appropriate meta-information that indicates that this version was being used to create a separate branch of the repository into a new one will be stored.

As it can be expected, no kind of operations with the branch of the repository will affect the original one. Branches have to be used, for instance, in cases when the development of an ontology have to be split into two separate processes or threads. A typical situation when this is necessary is when an old version has to be supported (which includes making small maintenance fixes) during the development of a new version. The state, which is used in production, can be branched and the development of the new one can take place in the branch while in the same time, the old version can still be supported and updated.

2.5.6 Controlling the History

*Those who control the past, control the future;
Those who control the future, control the present;
Those who control the present, control the past.*

1984, George Orwell

The possibility for tracking changes in a repository and gathering history has an important consequence: the history has to be controlled! The most important reasons for this and the appropriate mechanisms are discussed here.

Reason 1: *The volume of the data monotonously grows.* As far as nothing is really removed from the repository (the statements are only marked as “dead”) all the statements that were ever asserted in the repository together with their tracking and meta-information can be expected to be preserved forever. This way the volume of the data³ monotonously grows with each update.

Reason 2: *The history may need refinement.* The automatic tracking of the changes allows a great level of control, however the fine-grained information may also be obsolete or confusing. For instance, after the end of a phase of development of an ontology, the particular updates made in the process may be unimportant – often it is the case that finally the differences with a specific previous state (say, a version) are those that count.

Therefore the following levels of control over the tracking information will be implemented:

- *Clear the history before certain state* – in such case, all the statements that died before

² Actually those which are not “alive”.

³ The number of the statements and resources.

this state (say *s1*), together with any kind of tracking and meta-information about them will be permanently removed from the repository. The same applies for the labeled version before this state. All values of the update counter form a kind of “calendar” and all changes are logged against it. There will be two options for managing the tracking information for statements that were “born” before *S1* and died after it. Under the first option, they will be stated to be born and at a special moment “before the Calendar” – and all calendar records before *s1* will also be deleted. Under the second option, the “calendar” will be preserved, so no changes will be introduced to the tracking information for those statements that remain in the repository and the Calendar will be cleared from all the records that are not related to such statements.

- *Aggregate updates* – a number of sequential updates (say, between *UID1* and *UID2*) to be merged and made equivalent to specified one, say *UID3*. In this case, all references to *UIDs* between *UID1* and *UID2* will be replaced with *UID3*, which may or may not be equal to *UID1* and *UID2*.

2.6 Formal Representation of the Meta-Information

All the Knowledge Control System (KCS) related information would be represented in RDF according to a schema that could be found at: <http://www.ontotext.com/otk/2002/03/kcs.rdfs>. That includes tracking, versioning, and security information as well as custom user-defined meta-information. It is important to acknowledge that although this schema provides a well-structured conceptual view to the meta-information, its support by a repository may not be implemented directly after this schema because of obvious performance problems. So, the schema presents the way this information can be imported, exported, and accessed in RQL queries. It also facilitates good formal understanding of the supported model.

The basic idea is that all the meta-information is encoded via kind of special, easily distinguishable properties – namely such defined as sub-properties of a `kcs:metaInfo`. Also, all the related classes are defined as sub-classes of `kcs:KCSClass`. Here follows the set of pre-defined meta-properties, mostly related to the tracking information (the classes, together with those relevant to the security, could be found in sub-section 3.5.) The hierarchy of the properties is presented (in a proprietary format) together with the domain and range restrictions for each property:

metaInfo

```

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

```

The `bornAt` and `diedAt` properties define the lifetime of the statement via references to the specific updates. In similar manner we express the information associated with each particular update – the user who made it, the actual time and, etc.

Obvious extensions of the above schema are the Dublin Core primitives – there is no problem those to be declared to be sub-properties of `metaInfo`. The above-proposed model has number of advantages:

- Flexibility. Various types of meta-information could be defined – the appropriate schema has to be created with the only requirement the properties there to be defined as sub-properties of `metaInfo` and the classes as sub-classes of `MetaInfoClass`.
- The different meta-properties may have their appropriate domain and range restrictions.

- It is easy to “strip” or preserve the meta-info, just ignoring the statements which predicates are sub-properties of `metaInfo` and the resources of class `MetaInfoClass`.

2.6.1 Meta-Information for Statements

Assigning meta-information to statements is trickier than to resources at least because there are no URIs to identify each statement. For each statement that we like to attach a meta-information we need to apply a sort of explicit reification in order to associate the required meta-properties with the appropriate instance of `rdf:Statement`. A special class is defined in the KCS schema for this purpose:

```
<rdfs:Class rdf:about="&kcs;StatementMetaInfo"
  rdfs:label="StatementMetaInfo">
  <rdfs:comment>
    A common super-class for all the meta-information about statements
    in the KCS schema.
  </rdfs:comment>
  <rdfs:subClassOf rdf:resource="&rdf;Statement" />
  <rdfs:subClassOf rdf:resource="&kcs;MetaInfoClass" />
</rdfs:Class>
```

When we need to associate some meta-information to a statement in the repository we can instantiate `kcs:StatementMetaInfo` for that statement directly referring to its subject, predicate and object. Here is an example of such instantiation:

```
<kcs:StatementMetaInfo rdf:about="&metaex;status1" >
  <rdf:subject rdf:resource="&metaex;Jim" />
  <rdf:predicate rdf:resource="&metaex;childOf" />
  <rdf:object rdf:resource="&metaex;John" />
  <rdfs:comment>Comment on statement (Jim,childOf,John)</rdfs:comment>
  <metaex:customMetaProp>customMetaProp on statement (Jim,childOf,John)
  </metaex:customMetaProp>
</kcs:StatementMetaInfo>
```

In this case, two pieces of meta information are attached to the `<Jim, childOf, John >` statement. The first one is just a comment encoded using the standard `rdfs:comment` property. The second one is a sample for custom meta-property, which can be defined by the user. A full working⁴ example that demonstrates how meta-information for statements can be encoded is presented in: http://www.ontotext.com/otk/statement_metainfo_ex.rdf.

We can easily extract all the meta-information about specific statement using an RQL query. To do that we need the subject, predicate and object of the statement – it is sad but true, there is no other standard way to refer to or specify a triple. A sample query retrieving the meta-properties and comments about the statement `<Jim, childOf, John>` from the above example may look like:

⁴ When the KCS schema and the example are loaded in a SESAME repository, the queries presented in this sub-section really work and can be used to extract meta-information about the statements.

```

select
  @metaProp, result
from
  {X : $CX } &rdfs;subject {A},
  {X : $CX } &rdfs;predicate {B},
  {X : $CX } &rdfs;object {C},
  {X : $CX } @metaProp {result}
where
  A = &metaex;Jim and B = &metaex;childOf and C = &metaex;John and
  ( @metaProp = &rdfs;comment or @metaProp < &kcs;metaInfo ) and
  $CX > &rdf;Statement

```

The above is a bit simplified syntax; the following replacements should take place to make it real:

```

&rdf; -> http://www.w3.org/1999/02/22-rdf-syntax-ns#
&rdfs; -> http://www.w3.org/2000/01/rdf-schema#
&kcs; -> http://www.ontotext.com/otk/2002/03/kcs.rdfs#
&metaex; -> http://www.ontotext.com/otk/statement_metainfo_ex.rdf#

```

The result of that query over a repository containing the example mentioned above is:

@metaProp	Result
http://www.w3.org/2000/01/rdf-schema#comment	"Comment on statement (Jim,childOf,John)"
http://www.ontotext.com/otk/statement_metainfo_ex.rdf - customMetaProp	"customMetaProp on statement (Jim,childOf,John)"

The main idea of using a specific class as base for all the meta-information within the repository is to make it possible to implement it in the more effective way. All the instances will be organized and stored in the most appropriate way, probably, in separated tables into the database and when we need them, we can easily emulate them as ‘real’ statements.

2.6.2 Low-level Representation of Meta-Information for Statements

Although conceptually clear, the above model for keeping meta-information (including tracking data) for statements has at least the following problems:

- Technically speaking, it requires reification, which is the most criticized RDF(S) feature, also not supported by many tools;
- For each statement of “real” data, there should be five statements tracking it (one for each of the sub-properties of `kcs:trackingInfo` and three more that define the appropriate updates), i.e. the volume of the tracking data is five times (!) bigger than the volume of the repository without it.

In order to resolve this issues, the KCS meta-information will be actually stored and queried internally using more appropriate encoding – the RDF(S) engine will take care to support some level of mimicry to preserve the abstraction of the above presented schema for the applications. For instance, if the RDF(S) repository works on top of an RDBMS and there is a table each row of which represents a single statement, so, few more columns will be added to keep references to the tables with security and tracking information. This way the problems with the volume and performance will be resolved at least with respect to the most critical use cases.

Here follows a simplified sample of a possible relational structure and representation of the data (resembling a statement form the example in sub-section 2.5):

Statements							Updates		
SID	Subject	Predicate	Object	Born UID	Died UID	Lock	UID	Time	User
...
101	Ref_A	Ref_r2	Ref_E	6	8	Ref_U5	6	...	Ref_U3
102	Ref_B	Ref_r3	Ref_E	4	10	...	7	...	Ref_U5
103	Ref_A	Ref_r1	Ref_B	6	9	...	8	...	Ref_U3
104	Ref_D	Ref_r1	Ref_F	7	11	...	9	...	Ref_U6
...

The Updates table keeps the information relevant to each update: the time when it happened and the user who performed the update (this information can easily be extended on demand.) In the Statements table, for each statement the UID when it appeared and disappeared is kept as a reference to the Updates table.

With respect to the tracking of the changes, design as the one proposed above has a number of advantages compared to a variant where the update information is kept directly in the Statements table:

- All the necessary information is kept;
- The tracking information is not messing with the “real” information about the statements, however when necessary the two tables can be easily joined;
- The most basic operations (that are expected to be performed most frequently) can be done over the Statements table without need for joining with the Updates table. Such operation is to take all the statements that were “alive” at certain state identified by UID;
- There is significant reduction of the volume of the tracking information in cases of batch updates when multiple statements are getting added or removed at once and thus refer to one and the same UID.

3 Security and Access Control

*Ищу я выход из ворот,
но нет – сюда есть только вход
и то не тот*

Владимир Высоцкий

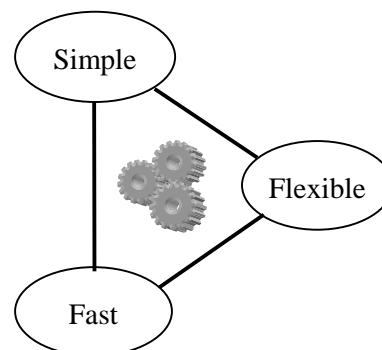
Here we define a model for access control over RDF(S) repositories. As argued in sub-section 1.3 this scope allows development of a system that handles ontologies, instance data, and knowledge bases of various types in the most unrestrictive way. The requirements for such security system are outlined first, followed by proposal for a model to satisfy them. The formal representation of the security data together with the implementation approach is discussed in the last sub-section.

3.1 Requirements

The requirements against a security model for an enterprise Knowledge Control System (KCS) are outlined. More general discussion of the context (environment, tasks, terminology) in which the access control should be thought and defined can be found in sub-section 1.3.

- (i) To be universal and flexible so to be suitable for wide range of domains, applications, enterprises and usage scenarios;
- (ii) To be applicable for both RDF(S) and DAML+OIL;
- (iii) To allow definition of comprehensive business logic (discussed in a sub-section below);
- (iv) To be simple and intuitive. As far as simplicity usually contradicts with the flexibility and performance, a minimal requirement would be to have a scheme that can be easily understood via comparison to some popular or well-studied and documented schemata. This requires at least well-understood notions as (groups, users, rights) to be used;
- (v) To be easy for implementation and verification;
- (vi) To degrade the performance as less as possible. Actually this is requirement for the implementation of the security system but it obviously depends on the scheme;
- (vii) To provide native support for restrictions which are typical for knowledge management and engineering tasks. Such are restrictions over the ontology (or schema) part of the repository, over instances of certain classes, over relation (or properties) of certain types;
- (viii) To allow streaming of the operations, say, generating, filtering, and sending back to the requester a big result set continuously, without waiting the whole set to be generated.

For the sake of expectations management, it has to be acknowledged that the above requirements are contradictory as proven by the history of development of various information systems, databases, operating systems, networks, etc. Flexibility and universality require compromises with both the performance impact of the access control and the simplicity. As presented in a simplified form on the “impossible triangle” diagram, there is no security system that is in the same time flexible, fast, and simple.



3.2 Related Work

There is almost no related work considering directly access control systems for knowledge bases, ontology management systems or repositories. In [Das et al, 2001] the Ontology Builder and Ontology Server products are presented, with short discussion on the security system included. It

becomes clear that the access rights there are defined in terms of roles assigned to the users, where each role provides permissions for certain operations. The so-called fine-grained permissions of the Ontology Server allow single user to have different roles with respect to different ontologies – which is the equivalent of the repository level security in our context.

Database security seems to be the area most closely related with the knowledge base security. On its turn it is a very broad field addressing wide variety of problems and specific needs. It is also the case that at the moment there are many different paradigms and approaches employed in this area. The standard security schemata provide control of the access down to the level of a database instance (repository) and distinct tables in a relational database. In contrast, we like our security system to provide means for controlling the access with much better granularity, namely at the level of instances or records.

An interesting exception presenting a more fine-grained access control down to record level is the Oracle Label Security presented in [ORACLE, 2001] where the reader can get a good impression on the underlying model as well as on the approaches and the mechanism underlying the implementation of such system.

Another relevant study dealing with the security issues for the databases accessed in a heterogeneous client-server environment (with accent on Internet) can be found in [Jones, 1996]. A comprehensive list of possible features of a security system is discussed there and although the author is not focusing on the record level access rights, some interesting analysis on the properties of systems supporting it can be found there.

A general discipline called role-based security covers wide range of theoretical issues related to the design and features of various access control system. As in the Flexible Authentication Framework (FAF) presented in [Jajodia, 2001], such systems can be described and studied in the following terms:

- *Data Items.* In general, any authorization framework determines the circumstances under which a user may attempt to execute an access operation on a given data item. The data items in our case are RDF resources and statements.
- *Access Types.* It is assumed that there exists some set of *actions* or operations that the user tries to execute on different data objects. In the case of an RDF(S) repository, such operations are the primitive operations like adding or removing statements, managing the security, various operations with the whole repository (like clearing) can also be distinguished.
- *Users and User Groups.* In general, when describing authorizations, we assume that there is some set of users, as well as groups consisting of such users. The user groups may or may not be organized in a hierarchy
- *Roles.* It is very common for users to assume roles – those can be seen as profiles or mappings between the various possible access rules and the users. It is question of choice for a security system whether roles to be supported at all – is it important to allow single user to play various roles, instead of just making a number of different accounts for him. It also depends on the specific application whether hierarchies of roles are supported or not.
- *Administration.* Any FAF must include an administrative policy that regulates who can grant authorizations and revoke them.

Related theoretical studies on the access control systems are also presented in [Schneider, 1998] and [Wijesekera and Jajodia, 2001]. Although these provide a lot of guidance for the design of a security system there are also aspects not covered there. A typical difference is that such systems consider data items with relatively simple and static properties. So, the security enforcement algorithms proposed are not directly relevant to the case of an RDF(S) repository where the objects represented has a quite complex descriptions and there properties may be altered via almost any change in the repository, i.e. the dependencies are hardly traceable.

3.3 Fine-grained Access Rights Support Business Logics Definition

In order a security system to support encoding of business logics it should be possible that comprehensive rules be defined to specify rights for specific portions and aspects of information to users or groups of users. Few examples of business logic rules follow:

Example 1: If there is document D1 in a repository, it should be possible that specific rights on this document be defined for different groups of users – some of them will have "need to know" for it, others will also be allowed to modify it.

Example 2: It may also be the case that there are types (or classes or categories) of documents that are visible just for a group of users – such example could be the document class "Minutes of Board Meetings".

Example 3: Again with respect to a document within a repository, different users may need to have different rights with respect to its meta-properties – there will be users that, for instance, will not be able to change the LevelOfConfidence property.

Example 4: In a skills management knowledge base (say, under the scheme presented in subsection 1.4) it should be possible to define that a user is able to modify the information about the skills of his subordinates but not to his boss.

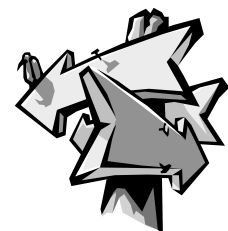
Many more examples for bread-and-butter business rules can be provided, but few basic requirements become obvious even from the above ones. The general requirement (iii) above can be extended with the following:

- (iii).R1. To be possible to assign rights to **certain classes** of objects or resources (see Example 2);
- (iii).R2. To be possible to assign rights to **specific objects** or resources (see Example 1);
- (iii).R3. To be possible to assign rights to **properties** or relations of specific types (see Example 3);
- (iii).R4. To be possible to assign rights **according to rules** that involve values of properties of objects, as well, as combinations and expressions between such (see Example 4).

It is very important to realize that the DBMS typically support access permission just to the level of granularity of table (thus partially covering requirement R1) and does not provide any support for rights say at the level of specific rows (requirement R2) or more complex dependencies (requirement R3). Most often in enterprise information systems the real security policy and business logic support is developed from scratch. According to the needs of the enterprise, the resources dedicated to this task, and the experience of the developers, specific information relevant to the security model is stored in the database (with or without relation to the access control features of the DBMS) and than interpreted by software modules where the appropriate logic is hard-coded in a way.

In the security model for a knowledge base or repository the above requirements are expected to be met. Such systems already have support for more comprehensive rules and dependencies between the objects – this is everything they provide on top of the databases. The knowledge-based systems are also expected to be self-sufficient with respect to the various types of data and rules that may be necessary for specific application or a multi-purpose enterprise system. It would be inefficient in general if there is some domain specific knowledge in the repository and the appropriate reasoning services available on top of it, but those may not be used to enforce the security policy and the business logic of each application.

Because of the above and following our intuition we claim that a knowledge-based repository should support fine-grained access control, at least satisfying requirements R1-R3. This understanding is built into the security model proposed in this section. There are two immediate consequences:



- Such repository will be not just “smarter” than a database, it will be more functional undertaking many of the functions currently performed by application servers or specific server-side modules;
- The design and implementation of such a security system is much more challenging because of the need bigger multitude and more complex security information to be managed.

The model proposed provides a kind of layered complexity that allows “management” of the compromise between the expressivity of the rules, the level of support, and the performance of the system. Supporting requirement R4 is possible on the highest level, Query restrictions.

3.4 Basic Principles of the Security Model

The basic principles underlying a security model for RDF(S) repositories that satisfy the requirements form the previous sections are presented here. Those in a high degree determine also the implementation approach and the representation of the security information discussed in the next sections.

***SPR1:** Access rights can be defined within an **RDF repository**.*

There may not be rules that simultaneously determine the rights for access to statements in multiple repositories. Of course a security schema created once for one repository can be copied (i.e. branched,) so, to be used as a basis for the access control in another one.

***SPR2:** Access to a repository is allowed only for registered **users**.*

At least ID and Password are kept for each registered user. One and the same user can have access rights with respect to multiple repositories. However, the user’s rights toward different repositories are relevant only for the repository they are defined for. Each request for operation on a repository is handled on behalf of a specific user.

***SPR3:** The following **restriction types** to be supported, according to the data they describe and the way they are defined:*

- **Repository** - the whole repository. Certain rights are applicable only for this restriction type (such as Admin and History). No need of definition as far as it is always used in the context of a repository;
- **Schema** – all the resources and statements that constitute the schema of the repository. No need of definition;
- **Classes** – all the resources (instances) of specific classes, including the statements where those are subjects. Disjunction logic applicable in case of multiple classes. Resources that are instances of sub-classes also considered. Defined via set of classes;
- **Instances** – set of specific resources, including the statements where those are subjects. Defined via set of resources;
- **Properties** – all the statements with specific properties as predicates. Disjunction logic applicable in case of multiple properties specified. The sub-properties are also considered. Defined via set of properties;
- **Pattern** – all the statements that conform to patterns defined via restrictions of type Classes or Instances over the subject and/or object and restriction of type Properties over the predicate;
- **Query** – the set of statements to be returned by RQL query that could take the current user as a parameter. Defined via query;

The so-called Security Classes provide additional power hooked to restriction type *Classes* – they are discussed in sub-section 3.4.1. A more general discussion on the different restriction types and how they should be supported can be found in sub-section 3.4.3.

SPR4: *The following Rights are supported: Read, Add, Remove, Admin, Clear, and History;*

Each right corresponds to a type of operation (or action) that is allowed or disallowed for user group with respect to some resources and statements. The case with the *Add* rights is more special because (in contrast to *Read* and *Remove* rights) it targets resources and statements, which are still not in the repository⁵ – this requires special handling in cases of Query restrictions or Security Classes involved.

Locking is not specified as a separate right because it can be considered as a consequence of the right for modification; a possibility to assign it separately for most of the cases will be just unnecessary complication and source for inconsistencies. There are two kinds of modifications possible⁶: Add and Remove. *Add* right is irrelevant for locking – you cannot lock something that is not in the repository yet. Following the above arguments, the locking is considered here as a consequence of the *Remove* right – the user can lock certain data iff s/he has a *Remove* rights for it.

There are also no separately defined rights for import and export – those are consequence of the *Add* and *Read* rights. If there is a separate repository-level right for import this may contradict with *Add* rights, because a file to be imported may contain any kind of statements and resources and particularly such not covered by the *Add* rights of the user. So, depending on whether the hypothetical import right or the *Add* right has higher priority, different paradoxes are possible. Similarly, if there is a specific right for export, it can easily cause inconsistency with the *Read* permissions.

The *History* right allows management of the tracking information; this includes labeling versions and the kind of modifications discussed in sub-section 2.5.6.

As mentioned above, *Admin* and *History* rights are supported just on the level of whole repositories – there is no support for admission of such rights for specific statements and resources.

SPR5: *Rights are always granted via Security Rules that have the following constituents: Restriction and Set of rights.*

Each security rule grants certain rights (out of those specified in SPR4) to the resources and statements determined by a single restriction, formed according to SPR3.

SPR6: *Roles are supported. Each role is defined as a set of Security Rules and other roles.*

The roles represent an abstraction that allows set of logically or otherwise related security rules to be grouped together. They are defined here in a way pretty close to the same notion in [Jajodia, 2001]. If role R1 is included in the definition of role R2, than the later (R2) indirectly contains also the security rules of the former (R1). This way, the roles may form a hierarchy with multiple-inheritance and unrestricted depth. Cyclic dependencies between roles are not allowed, of course.

SPR7: *Roles as well as individual Security Rules can be assigned to Users.*

A number of roles and individual rules can be assigned to registered users. Assigning a role is equivalent to assigning all the security rules and other roles that form its definition.

SPR8: *The users have only the rights explicitly assigned to them, i.e. a permissive security policy is enforced.*

⁵ Thanks to Michel Klein, who raised this issue within a discussion during the early analysis phase.

⁶ See sub-section 2.3 for comments on types of updates in an RDF(S) repository.

Apart from the exception defined with SPR9, the users are only allowed to perform actions for which a formal permission is assigned to them. A user is allowed to perform certain action over certain data iff there is at least one rule out of those (directly or via roles) assigned to him to grant this permission.

SPR9: *Each user can Read and Remove statements added by him.*

An exception to the permissive security policy is that the users can always see and remove statements they added (so, in a sense they “own”.)

SPR10: *Statements defining security rules, roles, and assignments are subject to Read, Add, and Remove rights of users that have Admin right for the repository.*

The above principle defines the administration policy (see [Jajodia, 2001]) in the terms of the security model introduced here.

3.4.1 Security Classes

Security classes are defined via RQL queries that return sets of resources⁷. Those allow specification of sets of resources that correspond to complex criteria (or definition) for which *Patterns* restrictions are not expressive enough.

Let us take for example the skills knowledge base presented in sub-section 1.4. If a group of users should be allowed to add and delete only skills for users of certain (class of) positions, the resources and statements to be considered can not be described like a pattern⁸ of existing classes and properties. A new security class, say, `sc1` that contains the above-specified skills have to be defined with the following RQL query:

```
select Y
from {X} #hasSkill {Y}, {X} #hasPosition {Z : #SomeTypeOfPosition}
```

Then a regular Class restriction can be applied on class `sc1`.

The security classes are not like the regular once. In RDF, particular resource `r1` is an instance of particular class `c1` iff there exists a statement `<r1, type, c1>` or equivalent one for a sub-class of `c1`. The intention behind the security classes is a bit different:

- They are defined as regular classes, i.e. of type `rdfs:Class`;
- There is a defining RQL query associated with each security class;
- Each resource is an instance of a security class iff it is member of the result of its defining query.

Obviously, the “security class” notion deviates from the class notion in RDF(S), in fact it is a bit similar to the notion accepted in the description logics and DAML+OIL. Important specifics about the implementation and the level of support for such security classes are discussed in sub-section 3.5.1.

3.4.2 Query Restrictions

The Query restrictions can bear unrestricted complexity including context-related logic that depends on the user making the request, i.e. the user for which the access rights are being evaluated.

⁷ An RQL query (analogously to SQL) may return a table with multiple columns as a result set, each row of which representing a single result. Only queries that return a table with single column as a result set can define security classes. For details on RQL see [Broekstra and Kampman, 2001a].

⁸ Neither it can be described as a set of patterns, because those may not be linked to each other or combined in another comprehensive way.

Technically speaking such restrictions are expressed via RQL queries that:

- Return RDF triples, subset of all the statements in the repository;
- Can involve `_USER_` parameter, to be replaced run-time with the user performing the request.

As discussed above, the security rule defined via *Query* restrictions hold on the statements returned from the evaluation of the query. In other words, in order to check whether single statement fits in the scope of rule defined via such restriction, the query has to be evaluated.

Such restriction could be used to meet the requirement (iii).R4 from sub-section 3.3 and manage cases as the fourth example presented there:

Example 4: In a skills management knowledge base (under the scheme presented in sub-section 1.4) to define that the user is able to modify the information about the skills of his subordinates but not to his boss.

Let us imagine that, in addition to the scheme we already presented, there is a property `subordinateOf` that relates a person⁹ with his/her boss. In such case, the business logic of the example could be encoded via security rule for Add and Delete rights using a *Query* restriction defined as follows:

```
select X, #hasSkill, Y
from {X} #hasSkill {Y}, {X} #subordinateOf { _USER_ }
```

The above query will return only those `<x, #hasSkill, y>` triples for which `x` is subordinate of the current user. Although the business logic is a bit simplified in the above example, it correctly illustrates the potential of the *Query* restrictions.

3.4.3 Three Layers of Complexity and Support

The restrictions to be used in the security system presented above can be separated into three levels of complexity:

- *Standard.* Restrictions that allow easy and fast implementation with small decay in the performance of the RDF(S) repository as compared to a system without any kind of access control. Those are all the restriction types without *Query* and without security classes involved. Details related to a possible implementation approach are presented in sub-section 3.5.1;
- *Extended.* Restrictions that involve security classes. The performance decay with respect to the standard requests to the repository is the same as for restrictions on the Standard level. However, a system that supports security classes should perform additional tasks under some strategy. Also, in a sense, those restrictions are not as instant as those on the Standard level. This becomes clear from the implementation approach proposed in sub-section 3.5.2;
- *Unrestricted.* Restrictions of type *Query*. Those restrictions require a separate query evaluation each time they have to be checked. They are definitely the computationally heaviest restrictions causing unpredictable performance decay. In the same time they allow virtually any type of business logic to be encoded and enforced instantly. Such can be used instead of security classes in cases when instant enforcement is important, as far as any *Extended* restriction could be expressed also as *Unrestricted* one. More details and appropriate implementation approach are discussed in sub-section 3.5.3.

⁹ It is assumed here that the same object (i.e. resource) represents the person in the subordination hierarchy as the user defined in the security system. Alternatively, there could be separate class (say *Employee*) in the skill management system and property that relates its instances to the users in the security system, then the above query should be complicated with one more join over this property.

The three-layer approach allows efficient management of the compromise between performance and comprehensive security policy. It is expected that in a well-designed security policy, most of the rules will be defined via *Standard* restrictions. Next, many of the complex but not critical or dynamic rules will be defined via *Extended* restrictions. Finally, most of the applications are expected to use just few or none *Unrestricted* rules that actually incorporate the power and complexity typical for the application servers inside the repository. In case such rules are really necessary, the performance the system depends on the complexity of the business logic and it should be compared to what usually comes as two separate layers – application server on top of a database or repository.

3.5 Implementation Approach

The implementation approach for the different aspects of the security enforcement is discussed in the next sub-sections. Finally, the schema for the formal representation of the security information is presented.

3.5.1 Standard Security Restrictions

The security model is carefully designed so to allow efficient implementation under the following assumptions:

- The schema taxonomies (both classes and properties) can be kept in the memory;
- All the security information can be kept in memory. This should be possible at least for the currently active users – those for which there are sessions opened. For users that do not use sessions, because of some reasons (say, the protocol used) a simple caching strategy should be possible;
- It is possible, without serious performance effect, to get the direct classes for the resources. In a relational DB storage implementation this could be implemented via separate table that takes care to store `rdf:type` statements or via special index over the table(s) keeping the statements;
- It is possible, without serious performance effect, the owner of each statement (i.e. the user, who added it) to be retrieved.

The following steps for handling of a Read request are possible:

1. Authenticate the user, if new session is being created or sessions are not applicable. Load the security information for the user, if it is not available in the memory. Process the roles and rules assigned to the user, to form a set of security rules applicable.
2. Perform the appropriate queries so to retrieve the requested data in the same way as if there is no security enforced. As additional requirements, retrieve also the information about the classes to which the resources included in the result belong as well as the owners of the statements.
3. Filter the result in the memory. Suppose the result is a stream of resources and/or statements to be filtered. For each statement or resource check does it fits in the restriction of at least one of the rules where the appropriate right is granted¹⁰, in this case Read. For the different types of restrictions it can be done as follows:
 - *Repository* and *Schema* restrictions – easy or no checks are necessary;
 - *Classes* restriction (say, on set of classes `CR1`) – take the classes to which the resource being checked directly belongs, let's name the set `RC`. Than check for a

¹⁰ It is obvious that rules granting only Write permissions are irrelevant for Read request and vice versa, so, the restrictions of such rules should not be evaluated against the result stream.

class in \mathbf{RC} that is member of $\mathbf{CR1}$ or sub-class¹¹ of a member of $\mathbf{CR1}$ – the resource is qualified under this restriction iff there is such class. If statement is being checked, its subject resource has to be checked under this procedure;

- *Instances* restriction (say, on set of instances $\mathbf{IR1}$) – the resource being checked qualifies under this restriction iff it is a member of $\mathbf{IR1}$. If statement is being checked, its subject resource has to be checked under this procedure;
- *Properties* restriction (say, on set of properties $\mathbf{PR1}$) – the statement being checked qualifies under this restriction iff its predicate \mathbf{P} is a member of $\mathbf{PR1}$ or sub-property of a member of $\mathbf{PR1}$.
- *Pattern* restriction – a statement qualifies to such restriction iff all the specified restrictions on the subject, predicate, and object are met. For a *Pattern* restriction there could be maximum three restrictions (one for each position in the triple) that belong to one of the above-discussed types, so, can be handled appropriately;
- Query restriction – discussed in sub-section 3.5.3 below.

The above described procedure allows streaming and requires two significant (as time-consumption) complications to the work of the repository (i) retrieving of the classes for all the resources being retrieved and (ii) checking sub-class and sub-property relations. The former is easy to support in a relational database with small amount of auxiliary data and/or indices and almost no the performance decay. The later requires in-memory taxonomy traversal that works considerably fast even for huge hierarchies.

Principle SPR9 has to be also considered in the above procedure. The first think to check when a statement is being filtered during Read and Remove operations, is whether the user who performs the request is the owner of the statement. If so, the statement should automatically pass the security check and no rules (i.e. restrictions) have to be evaluated towards it.

3.5.2 Security Classes Support

Security classes can be used in restrictions of type *Classes*. Each security class is defined via RQL query (called “defining query”) that returns resources. The strategy for support of such classes is to evaluate them periodically according to a customizable schedule. As a result, in each moment for each resource \mathbf{R} in the repository that was evaluated to “belong” to certain security class $\mathbf{SC1}$ (during the last evaluation for $\mathbf{SC1}$), this fact is explicitly “known” in the repository as if the statement $\langle \mathbf{R}, \mathbf{rdf:type}, \mathbf{SC1} \rangle$ was asserted. This way, the procedure for checking restrictions described above remains unchanged – it treats the security classes as the usual ones.

Of course, there are two important differences between the security classes and the regular ones:

- The support for security classes requires their defining queries to be evaluated from time to time and the appropriate statements to be asserted, so, the result to be kept in explicit form;
- It is possible the explicit knowledge whether certain resource belongs to a security class to be out of date. Due to a change in the repository it could happen that certain resource should not belong to the security class anymore, while another resource should “join” the class. This inaccuracy will be fixed with the next evaluation of the defining query, but before this wrong security judgments are possible.

Due to the above specifics, the security classes should be used carefully with the proper understanding of the administrator about the compromises being made. In the same time, for relatively static repositories or types of data the benefits are obvious – complex restrictions can be

¹¹ As far as the schema is kept in the memory, sub-class checking is easy. Same for the sub-property checks necessary for the *Properties* restriction type.

enforced with as low performance impact as for the standard ones.

The “update” procedure to be executed for each security class according to its schedule should: evaluate its defining query, remove the “instantiation” statements about resources that should be “disjoined” from the class, and add the appropriate statements for resources that should “join” the security class. This procedure should be carefully synchronized, so to disturb the “static” members of the security class as less as possible.

3.5.3 Query Restriction Type Support

The evaluation of the *Query* restriction types is fairly simple – the query has to be evaluated when such restriction is part of a security rule that has to be evaluated (i.e. it grants rights relevant to the request being processed.) In the most common scenario of Read request, before performing the real query to the storage layer, the defining query (\mathfrak{RQ}) of the restriction should be evaluated. The statements in the results of the actual request qualify iff they belong to the result of \mathfrak{RQ} . This approach allows streaming of the operations. However, special strategies may be necessary in case of defining queries that result in huge result sets.

The evaluation of Query restrictions for Add rights could be a bit tricky, as far, as the statements to be evaluated are not present in the repository, so, they will definitely not be part of the results set of the defining query. A simple technique that could be used is to add one or more statements so to accomplish the request. Next, to evaluate the defining query of the restriction, and check, which of the most recently added statements qualify. Those, which do not qualify, to be removed (together with any tracking information.)

3.5.4 Formal Representation of the Security Information

All the data necessary for the knowledge control system (KCS) can be formally represented in RDF according to the schema <http://www.ontotext.com/otk/2002/03/kcs.rdfs>. It is important to realize that KCS-data may be kept natively in a different format due to efficiency or security reasons¹². The important point is that this data logically follows this schema and can be extracted in RDF(S) if necessary.

Bellow a snapshot of the class hierarchy is presented in a proprietary format. Under each class, indented, the following items are listed:

- Its properties represented with their name and range. Only those properties for which the class is explicitly asserted as their domain are listed. It has to be considered that for each class, all the properties given for its super-classes are also relevant.
- Its sub-classes.

¹² See sub-section 2.6.1 for such motivation and details on the representation of the tracking information.

The scheme follows:

```

KCSClass
  MetaInfoClass
    StatementMetaInfo - subClassOf (rdfs:Statement)
  Update:
    madeBy -> User
    madeOn -> rdfs:Literal
  Version:
    versionName -> rdfs:Literal
    labeledBy -> User
    stateLabeled -> Update
  User:
    userID -> rdfs:Literal
    password -> rdfs:Literal
    Person:
      firstName -> rdfs:Literal
      lastName -> rdfs:Literal
  Restriction
    RepositoryRestrction
    SchemaRestrction
    ResourceRestriction
      ClassesRestriction:
        includeClass -> rdfs:Class
      InstanceRestriction:
        includeResource -> rdfs:Resource
    StatementRestriction
      PropertiesRestriction:
        includeProperty -> rdf:Property
      PatternRestriction:
        subjectRestr -> ResourceRestriction
        predicateRestr -> PropertiesRestriction
        objectRestr -> ResourceRestriction
      QueryRestriction:
        restrOnQuery -> rdfs:Literal
  Role:
    roleName -> rdfs:Literal
    superRole -> Role
    includeRule -> SecurityRule
  SecurityRule:
    rightsGranted -> rdfs:Literal
    ruleRestriction -> Restriction
SecurityClass:
  restrOnQuery -> rdfs:Literal

```

4 Instance Reasoning in On-To-Knowledge

The research, analysis, and the implementation approach for task “11.3 Reasoning Enhancements” is discussed in this section. A DAML+OIL reasoner called BOR is specified including the exact services, interface, and expressiveness to be supported. The section starts with a bit more general discussion on the various types of reasoning, DAML+OIL and RDF(S).

4.1 Reasoning in On-To-Knowledge Project

The represented knowledge allows (potentially) infinite number of possible uses, but within a practical system mainly the typical usage cases are implemented efficiently. In our view within the On-To-Knowledge project such typical usage cases are: (1) Ontology development and maintenance and (2) Ontology use. Ontology development requires the following basic tasks to be supported:

- Checking whether a class definition is consistent by itself or with respect to a set of other class descriptions;
- Checking whether a given class definition is more general than another class definition;
- Construction of explicit hierarchy of class names on the base of their class definitions.

We call these reasoning tasks "**Terminological Reasoning**". Ontology use involves first an already developed ontology in which the classes are defined and (possibly) the relations between them are explicitly represented (after some terminological reasoning) and next instance data of much higher magnitude, say, thousands or millions of instances. This usage case requires task such as:

- Find the most specific classes that describe a partially specified instance;
- Find all instances in the dataset which are instances of a given class definition;
- More complex queries involving instance data. Such could be getting all pairs of instances related in a way;
- Checking the consistence of the instance data with respect to the ontology.

We call these reasoning tasks "**Instance Reasoning**".

4.1.1 Discussion on Architecture with Separate Storage and Reasoning

Due to a number of different reasons the infrastructure initially developed under the On-To-Knowledge project separates the ontology and data representation from the reasoning over them. The storage, management, and querying of ontologies and instances is handled by the system SESAME – an RDF(S) repository which supports the RQL language. If more expressive reasoning is necessary (in more expressive language like OIL or DAML+OIL) then the corresponding information should be sent to an external reasoner (say, the FaCT system) that processes it and returns the answer back. Although such solution is appealing in terms of reuse of existing tools and compliance, in our view it can hardly provide good level of performance and interoperability in cases when the ontology and/or the instance data is very large. There are two possible approaches for implementation of such architecture:

- (i) Only the relevant parts of the ontology and the instance data are sent to the external reasoner. Such solution minimizes the exchange overhead between the two system, but imposes the question how these relevant parts are determined. In general the problem of fragmentation of an ontology and/or instance data into non-interacting chunks can require already considerable amount of reasoning.
- (ii) The external reasoner to support its own copy of the ontology and/or instance data. In this case the reasoner duplicates a lot of the functions of the repository, one way or another

everything “known” to the repository should be passed to the reasoner.

4.1.2 Requirements Towards a Reasoning Service for On-To-Knowledge

Both approaches are in contradiction with the expectation for the SESAME usage – thousands of classes and millions of instances. Thus, the integration with external reasoner seems feasible only for small ontologies.

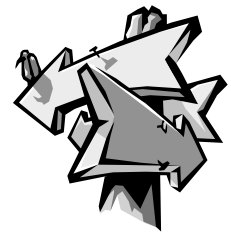
Using only SESAME is also not satisfactory solution because it supports only RDF(S) which expressive power is insufficient to support ontologies and instance data for domains and applications that require the full inventory of ontological languages such as DAML+OIL.

The above arguments motivated our position that the reasoning services in the project has to satisfy the following requirements:

- To be aware of the semantics of DAML+OIL
- To be efficient in the typical usage cases
 - Ontology development – Terminological Reasoning, usually no instances are involved;
 - Ontology use – Instance Reasoning, stable ontology with huge instance data.
- To be in close integration with the RDF(S) repository.

4.1.3 DAML+OIL Reasoner Built-In the Repository

In our view in order to satisfy the above stated requirements we have to implement the instance reasoning within the On-To-Knowledge project from scratch. The exact language and services to be supported are discussed in this chapter as well as the implementation approach. DAML+OIL represents an obvious choice for language to be supported – it is the latest successor of OIL, developed in cooperation with the DAML representatives under the W3C patronage with an objective to cover as many paradigms and applications as possible keeping in the same time the advantages that made OIL so popular.



DAML+OIL is shortly commented and compared to RDF(S) and the description logics in the following sub-sections. Next, the services to be supported are discussed together with analysis outlining possible algorithms for their implementation. Finally, the services to be supported are specified in terms of a high-level functional interface.

4.2 DAML+OIL and RDF(S)

DAML+OIL is a logical language whose syntax is built-in the RDF(S) and the semantics of the language is partially defined within the semantics of RDF(S) (for a definition of model-theoretical semantics of RDF(S) see [Hayes, 2001]). As a logical language DAML+OIL is furnished with a model-theoretical semantics presented in [Patel-Schneider, 2001]. Although this semantics is not a traditional one it is very close to the standard model-theoretical semantics in the classical logic. A sample¹³ DAML+OIL statement follows:

¹³ Taken from <http://www.daml.org/2001/03/daml+oil-ex.daml>

```

<daml:Class rdf:ID="Person">
  <rdfs:subClassOf rdf:resource="#Animal"/>

  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#hasParent"/>
      <daml:toClass rdf:resource="#Person"/>
    </daml:Restriction>
  </rdfs:subClassOf>

  <rdfs:subClassOf>
    <daml:Restriction daml:cardinality="1">
      <daml:onProperty rdf:resource="#hasFather"/>
    </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>

```

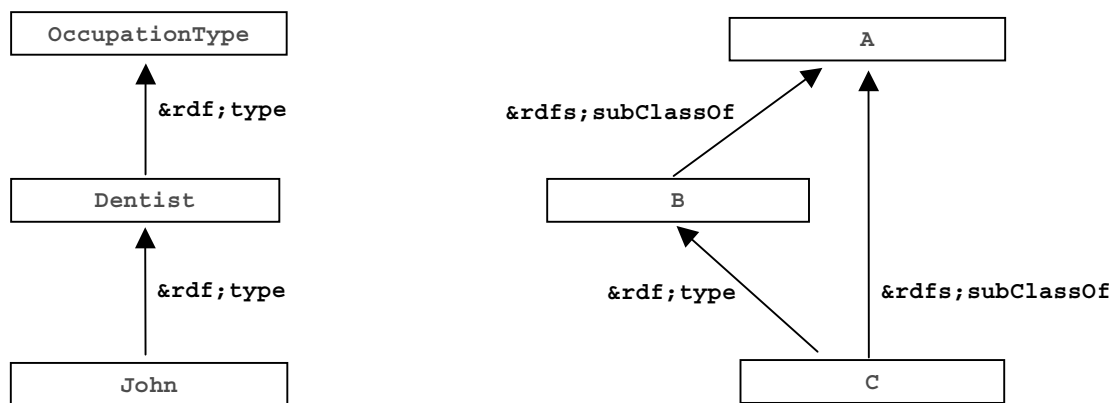
4.2.1 *The Incompatible Semantics*

There are certain elements of RDF(S) on which DAML+OIL depends (see [Patel-Schneider and van Harmelen, 2001]): (1) RDF triple structure and (2) RDF Schema constructions. Thus one can consider a DAML+OIL ontology as an RDF(S) ontology and can assign to it the corresponding RDF(S) semantics. But DAML+OIL has its own model-theoretical semantics which is different from the one of RDF(S) and ignores certain RDF(S) constructions and features: (1) reification, (2) containers, and (3) meta-classes. Thus we have a language which is situated within another language, but which separate itself from the semantics of the surrounding language.

Let us clarify the point from the previous paragraph. Reification is missing from DAML+OIL because the semantics of the language excludes the possibility the same resources to be used for both a class and a property. Container is a notion outside the definition of DAML+OIL ontology. Meta-classes are forbidden in DAML+OIL. Of course one can use the same resource as a class name and as instance name, but such resource couldn't receive the intended semantics in DAML+OIL. Thus real meta-class hierarchy is impossible in DAML+OIL because if a resource is used in a syntactic construction requiring an instance then this resource is always interpreted as singleton. Thus, if the same resource is also used as a class then it is very special class having only one instance. In our view this is not an advantage of the definition of DAML+OIL, but one unfortunate consequence of its definition. We offer a refinement of the definition of DAML+OIL ontology in the next section defining the notion of well-defined DAML+OIL ontology¹⁴.

A common misunderstanding is that DAML+OIL semantics is extension of the RDF(S) semantics. Although most of the primitives of the later have their cousins in the former, the formal interpretations are different. As mentioned earlier, meta-classes are not allowed in DAML+OIL. As a consequence DAML+OIL also does not support the flexible meta-modeling of RDF(S) (see [Pan and Horrocks, 2001] for details and criticism). Thus the following two examples are valid RDF(S) and although they represent syntactically well-formed DAML+OIL as well they are perfectly invalid there.

¹⁴ Actually, the term "DAML+OIL repository" will be introduced.



In the first case OccupationType appears to be a meta-class as far as its instance Dentist is a class itself (otherwise John may not be its instance.) The second example is a bit more complex – both B and C are classes, because they cannot be sub-classes of A otherwise. However it means that B is meta-class because C is its instance.

4.2.1.1 Stratified Meta-Modelling Schemata

The A-B-C example above is interesting also because it cannot be determined what is class and what is meta-class there. As far as B is a meta-class A should also be such because if B can have classes as instances and so does A, too. But than C should also be a meta-class, because it is a sub-class of A. It appears that that B should be meta-meta-class because its instance is a meta-class. Than A is a meta-meta-class as well and because of it C, too. This example is a good demonstration for the nature of RDF(S) – no kind of layers or strata can be defined there.

Let us call a meta-modeling scheme “stratified” if it does not allow one class to be also a meta-class. More formally speaking, to be possible all the resources to be split into strata (non-intersecting layers) so that each resource to be an instance of a resource from the intermediate upper strata. Example like John-Dentist-OccupationType would be valid under a stratified meta-modeling, while the A-B-C example will not be. It is an interesting question, can DAML+OIL be extended to support stratified meta-classes.

4.2.2 Incompatible Reasoning

The incompatibility between the DAML+OIL and RDF(S) semantics leads to principle problems with the compatibility of any reasoning services – each time when repository is interpreted and verified either one of the semantics should be use either the other. Although it is possible a sub-language of RDF(S) to be defined that is “forward-compatible” with DAML+OIL, there is no widely accepted dialect like this.

For the implementation of the DAML+OIL reasoner the strict semantic of the language will be used which means that the standard two-level model with disjoint sets of classes and instances will be supported. The appropriate distinctions are more precisely defined in the next sub-sections.

4.2.3 Separation of DAML+OIL from RDF(S)

For the purpose of using DAML+OIL within the On-To-Knowledge project we can say that SESAME is appropriate for storing DAML+OIL ontologies and datasets because they are RDF(S) constructs. However, additional facilities are needed to process DAML+OIL in proper way – such, corresponding to the semantics of DAML+OIL.

In the rest of section 4 we are concerned with DAML+OIL ontologies and instance data represented as RDF(S) repositories in SESAME. Our separation of DAML+OIL from RDF(S) is based on the propositions made in the following two citations:

[van Harmelen et al. 2001], "A *DAML+OIL* ontology consists of zero or more headers, followed by zero or more class elements, property elements, and instances." and

[Patel-Schneider, 2001], "A *DAML-OIL* knowledge base is a collection of *RDF* triples. Some of these triples are relevant to *DAML-OIL* and some are not. This semantics currently only treats the triples that are obviously relevant to *DAML-OIL* and ignores the rest."

Thus, we will try to classify all the resources and statements in a *SESAME* repository containing a *DAML+OIL* knowledge into several categories. The resources are classified as follows:

- **Class** – all object classes defined within some class element of the *DAML+OIL* ontology (including the imported ones which minimally contain "#Thing" or "#Nothing" the two *DAML+OIL* standard classes for the universal and absurd concept). This set of resources contains all *DAML+OIL* classes defined in the ontology;
- **Property** – all object or datatype properties defined within some property element of the *DAML+OIL* ontology (including the imported ones) and thus this set of resources contains all *DAML+OIL* properties defined in the ontology;
- **Instance** – all resources that are used as instances in *DAML+OIL* elements. They can be used inside class element (by *daml:hasValue* element) or by instance statements.
- **Others** – all resources used in the *DAML+OIL* ontology outside the headers which are not in the previous sets. Examples of such are the relational axioms.

In order one *RDF(S)* repository to be also a well-defined *DAML+OIL* repository we will require the sets *Class*, *Property*, and *Instance* to be pair-wise disjoint.

The statements in a *DAML+OIL* repository are classified into the following sets:

- **header** – containing all header statements;
- **ontology** – containing all class and property elements;
- **dataset** – containing all instance statements;
- **other** – containing all other statements.

Although the set 'other' is not necessary empty the statements in it don't have any impact on the *DAML+OIL* semantics of the ontology. The principle for classification of the statements is discussed in [Simov, 2002].

In the following text we will use the term 'Repository' to point to the *SESAME* *RDF(S)* repository which contains well-defined *DAML+OIL* repository. The inference(s) will be concerned with the ontology and the dataset parts of such a repository. When it is necessary a distinction between the two types of knowledge (class and property versus instance knowledge) to be made, we will use the term 'Ontology' for the ontological part of the repository and 'Dataset' for the instance part. All syntactic constructions defined in [van Harmelen et al. 2001] as elements are called in the following texts "elements." Instance statements are called instance statements. In places where both kind of syntactic constructions can be used we are using elements to cover *DAML+OIL* elements and instance statements. This has to be clear from the context.

4.3 *DAML+OIL* as a Description Logic

In this sub-section we discuss *DAML+OIL* as a kind of description logic in order to be able to reuse known algorithms and results on the tractability of such languages. The closest description logic known in the literature is *SHOQ(D)*, described in [Horrocks and Sattler, 2001].

The *SHOQ(D)* language primitives (without obvious constructs like subsumption) are presented below together with their *DAML+OIL* equivalents:

<i>SHOQ(D)</i>	DAML+OIL
Concepts	Classes
Roles	Properties
Individuals	Instances
Nominals	daml:hasValue
Concrete datatypes	Datatypes
Conjunction	daml:intersectionOf

<i>SHOQ(D)</i>	DAML+OIL
Disjunction	daml:unionOf
Negation	daml:complementOf
Exists restriction	daml:hasClass
Value restriction	daml:toClass
At-least restriction	daml:maxCardinality
At-most restriction	daml:minCardinality

Reverse role operator is missing in *SHOQ(D)* and thus (daml:inverseOf element) cannot be represented in *SHOQ(D)*. The rest of the primitives of DAML+OIL can be represented in *SHOQ(D)* as role inclusion axioms, transitivity axioms, generalized concept inclusion axioms, equality axioms, inequality axioms.

In [Horrocks and Sattler 2001] an inference procedure for terminological reasoning in *SHOQ(D)* is presented. An implementation of it is under development as an extension of FaCT system. Thus even with respect to the more expressive language DAML+OIL the original architecture of the division of labor between SESAME and FaCT is still possible.

4.4 Inference Services in Description Logics

The fact that DAML+OIL could be considered as a description logic allows us to use the extensive investigations of these logics. A knowledge representation and reasoning system based on DLs is assumed to provide the number of reasoning facilities that can be separated in two groups: terminological and instance reasoning.

4.4.1 Terminological Reasoning

The four tasks below are the basics of the terminological reasoning – they are implemented in most of the systems, including FaCT.

- **Subsumption check** – checks if one class is a subclass of another;
- **Consistency check** – checks for inconsistent class definitions;
- **Taxonomy construction** – computes all subclass relations (including those which are not explicitly stated but that are implied by the given definitions);
- **Classification** – determines the classes that immediate subsume or are subsumed by a given class;

4.4.2 Instance Reasoning

The basic two tasks involving individuals are:

- **Realisation** – given a partial description of an individual (instance), finds the most specific concept that describes it;
- **Instance checking** – given a partial description of an individual (instance) and a class description, finds whether the class describes the instance;
- **Individual retrieval** – finds all individuals (instances) that are described by a given concept;

In description logics with negation most terminological reasoning problems are reduced to (in)consistency of a class definition. The most widely used technique for the (in)consistency

checking is based on the notion of a **tableau**. On a very informal level a tableau is an abstract structure representing all explicit atomic knowledge about a set of instance variables (or names) following from a given ontology. A tableau is a base of construction of an interpretation in which the corresponding knowledge is satisfiable. Thus the decision procedure tries to show that a tableau corresponding to the current ontology and/or dataset exist. Sometimes the corresponding tableau is infinite and its existence is proven indirectly by construction of a completion. A **completion** is an abstract structure, which ensure the existence of a tableau. Instance Reasoning can be realized in a similar way but including the instance names from the dataset in the completion.

The reasoning procedure is designed as a set of rules. The rules are applied to an (incomplete) abstract structure explicating the knowledge stored in the structure and trying to construct a completion. The rules are classified in two different ways:

4.4.3 *Deterministic versus non-deterministic*

A deterministic rule modifies the current structure in one possible way. A non-deterministic rule could modify the structure in more than one way and thus leaves a choice to the system.

4.4.4 *Extending versus non-extending*

A rule is extending if its application introduce at least one new instance variable in the structure.

Obviously deterministic, non-extending rules are preferable in one implementation. The application of the rules is governed by meta-rules and control information:

A meta-rule could be as:

apply first the non-extending rules then extending ones

Control information could concern the typical problems that are expected and thus could require different meta-rules.

In the worst case for expressive languages like DAML+OIL (*SHOQ(D)*) the inference is very hard - usually **NP** and more. Thus in the systems usually some optimisations for the average case problems are implemented. Such optimisations includes heuristics to avoid “expensive” checks or incomplete inferences like *pseudo models* - sound, but incomplete inference based on structures similar to completions [Haarslev V., Mueller R., Turhan A.-Y., 2001].

In our work we will use all of these treasure of experience trying to adopt it to the typical cases of instance reasoning within the On-To-Knowledge project.

4.5 Instance Reasoning in On-To-Knowledge Project

In order to make the reasoning task feasible within the time constraints we offer one graduate approach to the tasks that we will implement. First we classify the possible datasets with respects to their complexity. On one end of the scale we put the datasets in which instance statements of an arbitrary complex form are presented. On the other end of the scale we put datasets which contain only ground instance statements. Such datasets are called ground datasets.

General instance statements

- a : C** - class statement: **C** is an arbitrary complex class expression
- (a,b) : R** - property statement: **R** is a property name
- a = b** - equality statement
- a ≠ b** - inequality statement
- ∀x.C(x)** - universal statement : **C** is an arbitrary complex class expression

a, b are instance names, **x** is an instance variable

The class and the universal statements provide high expressivity within the datasets. They allow one to state additional ontological restrictions over a particular dataset. Usually these statements impose great computational problems to the inference procedure.

Here are examples of such statements, $\mathbf{a} : (\mathbf{Mother} \cap (\geq 3 \text{ has-child}))$ states that \mathbf{a} is a mother with more than two children. $\forall \mathbf{x}. (\neg \mathbf{Mother} \cup (\geq 3 \text{ has-child}))(\mathbf{x})$ states that each mother in the current domain has more than two children.

Ground instance statements

$\mathbf{a} : \mathbf{A}$	- concept statement: \mathbf{A} is a class name
$(\mathbf{a}, \mathbf{b}) : \mathbf{R}$	- property statement: \mathbf{R} is a property name
$\mathbf{a} = \mathbf{b}$	- equality statement
$\mathbf{a} \neq \mathbf{b}$	- inequality statement

Our expectations are that datasets containing only ground instance statements will allow very efficient implementation of inference procedures. Also ground datasets are typical for the case studies within the On-To-Knowledge project. It is important to mention that universal statements are consequences from the ontological knowledge and thus they can be expected to be always presented as constraints over the instance data.

In the rest of this section we describe the inference services that we envisage to implement as an extension of SESAME system.

4.5.1 Realisation

The task can be formally defined as follows:

Given: \mathbf{A} - a set of instance statements, \mathbf{a} - an instance resource in \mathbf{A} , and $\mathbf{C}_1, \dots, \mathbf{C}_n$ - class resource in the ontology

Find: $\mathbf{C}_{i1}, \dots, \mathbf{C}_{im}$ the most specific classes that describe \mathbf{a}

The decision procedure can require consistency check of the augmented sets $\mathbf{A} \cup \{\mathbf{a} : \neg \mathbf{C}_j\}$.

Our goal here will be to minimise the number of the classes \mathbf{C}_j for which this check will be done. We will try to develop a special procedure for the case when \mathbf{A} is a set of ground instance statements.

4.5.2 Instance checking

The task can be formally defined as follows:

Given: \mathbf{a} - an instance statements, and \mathbf{C} - a class element

Find: whether \mathbf{a} is an instance of by \mathbf{C}

4.5.3 Retrieval

The task can be formally defined as follows:

Given: \mathbf{A} - a set of instance statements, and \mathbf{C} - a class element

Find: $\mathbf{a}_1, \dots, \mathbf{a}_m$ - the instance resources in \mathbf{A} that are instances of \mathbf{C}

The decision procedure can require consistency check of the augmented sets $\mathbf{A} \cup \{\mathbf{a} : \neg \mathbf{C}\}$ for all instances in \mathbf{A} .

Our goal here will be to minimise the number of the instances for which this check will be done.

4.5.4 Retrieval of components

Besides the individuals one can require retrieval of their components as well. A component of an instance consists of all instances connected to the given instance by some DAML+OIL property. This sort of inference is important when some of the instances in the dataset are represented by numbers or other kind of ids and the important information is given via the properties defined on these ids. Of course some restrictions on the size and content of component of instance have to be imposed in practice. The component retrieval is an open research problem.

4.5.5 Model Checking

The task can be formally defined as follows:

Given: **A** - a set of instance statements, and **O** - an ontology

Find: whether **A** is a model of **O**

Here one of the important thing is the information presented in **A**. Important assumption is that it already contains all information for a tableau and there is no need to add information. Only checking for consistency of the statements with ontology and between them. This means that we will not add any information to the dataset, but we will check whether it is a tableau.

We envisage this task to be efficiently decidable. Our goal will be to modify the completion rules to the restriction that no new instances can be added to the data. Special attention will be paid to set of ground instance statements.

This inference service is a new one for description logic world. It can be very useful for compatibility check between versions of ontologies and datasets.

4.5.6 Minimal Sub-Ontology Extraction

The task can be formally defined as follows:

Given: **A** - a set of instance statements, and **O** - an ontology

Find: ontology **O'** - a minimal sub-ontology with a model **A**

A minimal ontology is defined as a minimal sub-taxonomy, but also such ontology will need to include some non-hierarchical knowledge (because of generalised concept inclusion axioms).

This inference service is a new one for description logic world. It can be very useful for determination the scope of an ontology exchange, for example, when certain information (typically a set of instances) has to be exchanged between two systems (or databases or knowledge bases).

4.6 Implementation

A new reasoning module BOR¹⁵ will be developed as an extension of the SESAME RDF(S) repository. The computational complexity of the above-mentioned reasoning problems is too high in order to allow straightforward implementation. One direction for improvements of the inference is usage of different strategies of “compilation” of knowledge via explication of implicit relationships within ontologies. Let us call this approach “pre-reasoning”. Here the variations of experiments could range between full explication and query-based expansions. The first kind of processing expands as much as possible relationships and via indexing mechanism uses these new relationships during inference. This approach has a serious drawback – the required memory could be enormous. The second approach will expand only these implicit relationships that are used for the evaluation of the queries presented to the inference engine. This way, only local expansions

¹⁵ It is not an abbreviation, just means “pine” in Bulgarian.

will be necessary. Other approaches between these two can be implement different control strategies. At least the following strategies will be used:

- some inference results will be pre-reasoned and stored in SESAME together with the raw data. The most important tasks to be performed are taxonomy and instance classification (realisation). The results will be stored back in SESAME as standard RDF(S), so, all the SESAME clients will be able to benefit.
- on demand SESAME will call BOR for query expansion and retrieval (see below).

BOR will close an important gap in the current system architecture of On-to-Knowledge. Currently, the query engine support by SESAME cannot handle many aspects of DAML+OIL (define class definitions, cardinality constraints etc.) when answering queries over instances. BOR will integrate DL-based reasoning into query answering over instances. BOR can be considered as "latent" instance reasoner because for many purposes it will be working hidden behind SESAME. For example, the results of the pre-reasoning will be available in SESAME, so, "the user" will not directly "see" BOR - he/she will just work with the results of its background work.

Besides the development of the inferences discussed above, we will pay attention to the few issues discussed in the next subsections.

4.6.1 The structure of the ontology and instance representation

We envisage some additional structuring over the representation of an ontology and/or dataset in order to allow some inference problems to be implemented as simple table look-up, or some of the result of the expensive inferences to be stored for multiple uses. The structures will be designed so to support tasks like construction of taxonomy over the class names in the ontology or realisation of all instances and ordering the instances with respect to taxonomy over the class names in ontology.

4.6.2 The control over the inference process

We are planning to develop two kinds of control over the inference process. First, we will pay attention to control rules similar to the meta-rules mentioned in the section 4.4. These rules will depend on the state of the ontology and/or the dataset and they will have global scope in a sense that they will be applicable to any ontology and/or dataset. The second kind of control will depend on the tasks for which the current ontology and/or dataset will be used. Such control information will have to be supplied by the user. It will concern the expectations for some inference to succeed or to fail depending on the application domain.

4.6.3 Datatypes

DAML+OIL allows for an expressive definition of new datatypes to cover the datatypes defined in XML Scheme. In our implementation we will not allow such expressivity and we will use pre-implemented "reasoners" over some of the datatypes such as integers and strings and we will use them as oracles inside the instance reasoner. The architecture of the reasoner will allow oracles for new datatypes to be added on demand.

4.7 Functional Interfaces to a DAML+OIL Reasoner

In this section we propose a collection of interface functions necessary for effective use and maintenance of repositories of DAML+OIL represented knowledge that may contain both ontologies and dataset (instance data) as it was explained above. Here we consider the repositories as abstract objects and the corresponding interfaces are operations working on these abstract objects and producing new abstract objects. The interfaces are defined as functions over repositories and elements. The actual programming interfaces (API) are defined in the next section and they reflect the actual structure of the represented knowledge and the elements defining this

knowledge. We are working in the following with a current repository, but by an optional argument *Repository* we can point to a particular repository.

4.7.1 Tell Interfaces

Tell interfaces give possibilities for adding knowledge to the repository. The general definition of the tell interfaces is as follows:

$$\text{Tell}(\Delta, \alpha) = \Delta'$$

where Δ is the current state of the repository, α is a new element to be added to the repository, and Δ' is the new state of the repository.

Some of these interfaces don't add new information, but instead only initiate some processing over the repository. In this case α is a command for the operation which to be performed over the repository.

Typical interfaces in the description logics are:

`defconcept(concept_name,concept_term[,tbox]).`

`defrole(role_name,role_term[,tbox]).`

`classify([tbox]).`

The first two clauses are examples of the first kind of tell interfaces where new axioms are added to the terminological part of the knowledge. The last clause is an example of the Tell interfaces which only initiate processing over the repository.

As it was mentioned, in the case of DAML+OIL: "A DAML+OIL ontology consists of zero or more headers, followed by zero or more class elements, property elements, and instances." [van Harmelen et al. 2001]. The syntax of any of the elements of a repository is determining uniquely its category. Thus we will reduce the number of the interfaces because the actual elements are explicit about their meaning and there is no ambiguity. Also some of inference services described above will be defined here as tell interfaces similar to `classify()`. The interfaces are:

tell(*Element*[,*Repository*]) - returns *true* if the element is successfully added to the repository, otherwise *false*. The element could be rejected if it makes the repository inconsistent. This has to be an option. The repository is changed to a new state.

classify([*Repository*]) - doesn't return value. It initiates classification of the repository.

initialization([*Repository*]) - creates an empty repository.

initialisation([*Repository*]) - creates an empty repository. This is exactly the same as previous one but with different syntax.

4.7.2 Delete Interfaces

Delete interfaces allow deletion of elements from a repository. The general definition of the delete interfaces is similar to this of tell interfaces:

$$\text{Dell}(\Delta, \alpha) = \Delta'$$

where Δ is the current state of the repository, α is an element to be deleted from the repository, and Δ' is the new state of the repository. One requirement here is that α is a distinguished element in Δ .

One important remark here is that deletion of a element is not equivalent to the assertion of its negation. Thus, if we have

$$\text{Tell}(\Delta, \neg\alpha) = \Delta' \text{ and } \text{Dell}(\Delta, \alpha) = \Delta''$$

The ontologies Δ' and Δ'' could be very different.

Again here we assume that what is deleted is a DAML+OIL element and its interpretation is

unique within the repository. Only elements that were explicitly added to the repository by the user could be deleted. A great problem with deleting of elements is what to do with the elements that are added by the inference engine and depend of the deleted element. In general this is a problem of theory revision but here we will not be concerned with this problem.

delete(*Element*[,*Repository*]) - returns *true* if the element is successfully deleted from the repository, otherwise *false*. The repository is changed to a new state.

4.7.3 Ask Interfaces

Ask interfaces query the repository. The general definition of the ask interfaces is similar to this of tell interfaces:

$$\text{Ask}(\Delta, \alpha) = \beta$$

where Δ is the current state of the repository, α is an element representing the query, and β is the answer. β can be a DAML+OIL element or a set of such elements (sub-repository), but it also could be true or false answer, or a set (list) of resource names which to be of some kind: class, property, or instance resource.

In our definition we will follow the ideas of [Bechhofer et. al. 1999] and define the inference queries for retrieval of definitions of the elements of the ontology see next section. The interfaces are:

direct_supers_c(*Class*[,*Repository*]) - returns a list of class resources *ClassList* which are the direct super classes of *Class*.

all_supers_c(*Class*[,*Repository*]) - returns a list of class resources *ClassList* which are the super classes of *Class*.

direct_subs_c(*Class*[,*Repository*]) - returns a list of class resources *ClassList* which are the direct sub-classes of *Class*.

all_subs_c(*Class*[,*Repository*]) - returns a list of class resources *ClassList* which are the sub-classes of *Class*.

top_c([*Repository*]) - returns a list of the resources of the most general classes defined in the repository under "#Thing". This operation can be defined using other interfaces, but sometimes is more convenient to have it explicitly defined.

equivalent_c(*Class*[,*Repository*]) - returns a list of the classes equivalent to the *Class*.

direct_supers_r(*Property*[,*Repository*]) - returns a list of property resources *PropertyList* which are the direct super properties of *Property*.

all_supers_r(*Property*[,*Repository*]) - returns a list of property resources *PropertyList* which are the super properties of *Property*.

direct_subs_r(*Property*[,*Repository*]) - returns a list of property resources *PropertyList* which are the direct sub-properties of *Property*.

all_subs_r(*Property*[,*Repository*]) - returns a list of property resources *PropertyList* which are the sub-properties of *Property*.

taxonomy_position(*ClassElement*[,*Repository*]) - returns three lists of class resources: *ClassListUp*, *ClassListDown*, *ClassListEq* where *ClassListUp* contains the direct super classes of *ClassElement*, *ClassListDown* contains the direct sub-classes of *ClassElement*, and *ClassListEq* contains the classes that are equivalent to *ClassElement*.

satisfiable(*ClassElement*[,*Repository*]) - returns *true* if the *ClassElement* is satisfiable and *false* otherwise.

subsumes(*ClassElement1*,*ClassElement2*[,*Repository*]) - returns *true* if the *ClassElement1* is a

subclass of *ClassElement2* and *false* otherwise.

equivalent(*ClassElement1*,*ClassElement2*[,*Repository*]) - returns *true* if the *ClassElement1* is equivalent to *ClassElement2* and *false* otherwise.

consistent([*Repository*]) - returns *true* if the *Repository* has a model and *false* otherwise. Here different inferences will be performed on the base of the elements presented in the repository.

realisation(*InstanceList*[,*Repository*]) - returns a list of class resources *ClassListUp* containing the classes describing all of the instances in the *InstanceList* if such exist and empty list otherwise. This is a small generalisation over the realisation defined above, that corresponds to *InstanceList* containing only one element. If the list contains more than one element then the result is a list of the more specific classes that describe all instances in the list. Alternative spelling could be **realization**.

instance_check(*Instance*,*ClassElement*[,*Repository*]) - *Instance* is an instance resource, *ClassElement* is a DAML+OIL element. Checks whether the partial description of *Instance* is a subclass of *ClassElement*.

retrival(*ClassElement*[,*Repository*]) - returns a list of instance resources *InstanceList* containing the instances of *ClassElement* if such exist and empty list otherwise.

retrival_components(*ClassElement*[,*Repository*]) - returns a list of instances that are described by *ClassElement* and to each instance in the list a graph representing its component or a part of the component is attached. If *ClassElement* has no instances this function returns an empty list.

model_checking([*Repository*]) - returns *true* if the dataset part of the repository is a model of ontological part of it, otherwise *false*.

ontology_extraction([*Repository*]) - returns *NewOntology* - a list of minimal sub-ontologies with respect to the dataset part of the repository. This is a list of lists of DAML+OIL class and property elements. The information in the headers is not extracted.

4.7.4 Display Interfaces

Display interfaces allow examination of the resources and elements in the repository. The interfaces are:

classes([*Repository*]) - returns a list of all class resources used in the repository.

properties([*Repository*]) - returns a list of all property resources used in the repository.

instances([*Repository*]) - returns a list of all instance resources used in the repository.

literals([*Repository*]) - returns a list of all literals used in the repository.

resources([*Repository*]) - returns a list of all resources used in the repository.

class_elements([*Repository*]) - returns a list of all class elements in the repository.

property_elements([*Repository*]) - returns a list of all property elements in the repository.

instnce_statements([*Repository*]) - returns a list of all instance elements in the repository.

elements(*Constant*[,*Repository*]) - returns a list of all elements in the repository which use the *Constant*. The *Constant* could be a resource or a literal.

5 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 will be changed, which are the new and the modified modules. Next the components and the interfaces are discussed in more details.

5.1 The Initial Design and Why it Changed

Before the description of the proposed design, it should be mentioned that initially the OMM was expected to build on top (or more precisely around) the SESAME RDF(S) repository. During the analysis phase it appeared that a better approach would be to extend the SESAME, so to modify its architecture and implement the appropriate features either by modifications to existing modules either by adding new ones. There are number of reasons motivating this change:

- The SESAME architecture became more advanced (as compared to the time when the proposal was written) so now it already allows the necessary features to be implemented as parts of it. For instance, the multi-protocol client access is now possible through the Request Router (see the next section for details).
- The analysis showed that implementation within SESAME will allow much better performance and reduction of the volume of the “auxiliary” data especially with respect to the implementation of the change tracking.
- The architecture now is flexible enough to allow plugging of new modules, namely storage and inference layers.

In this situation it is obviously inefficient to implement OMM that is building a new architecture around SESAME using it just as a basic RDF(S) storage. It is also the case, that the existing tools using SESAME will easily start using the new features if they come as its extension rather than as a new module.

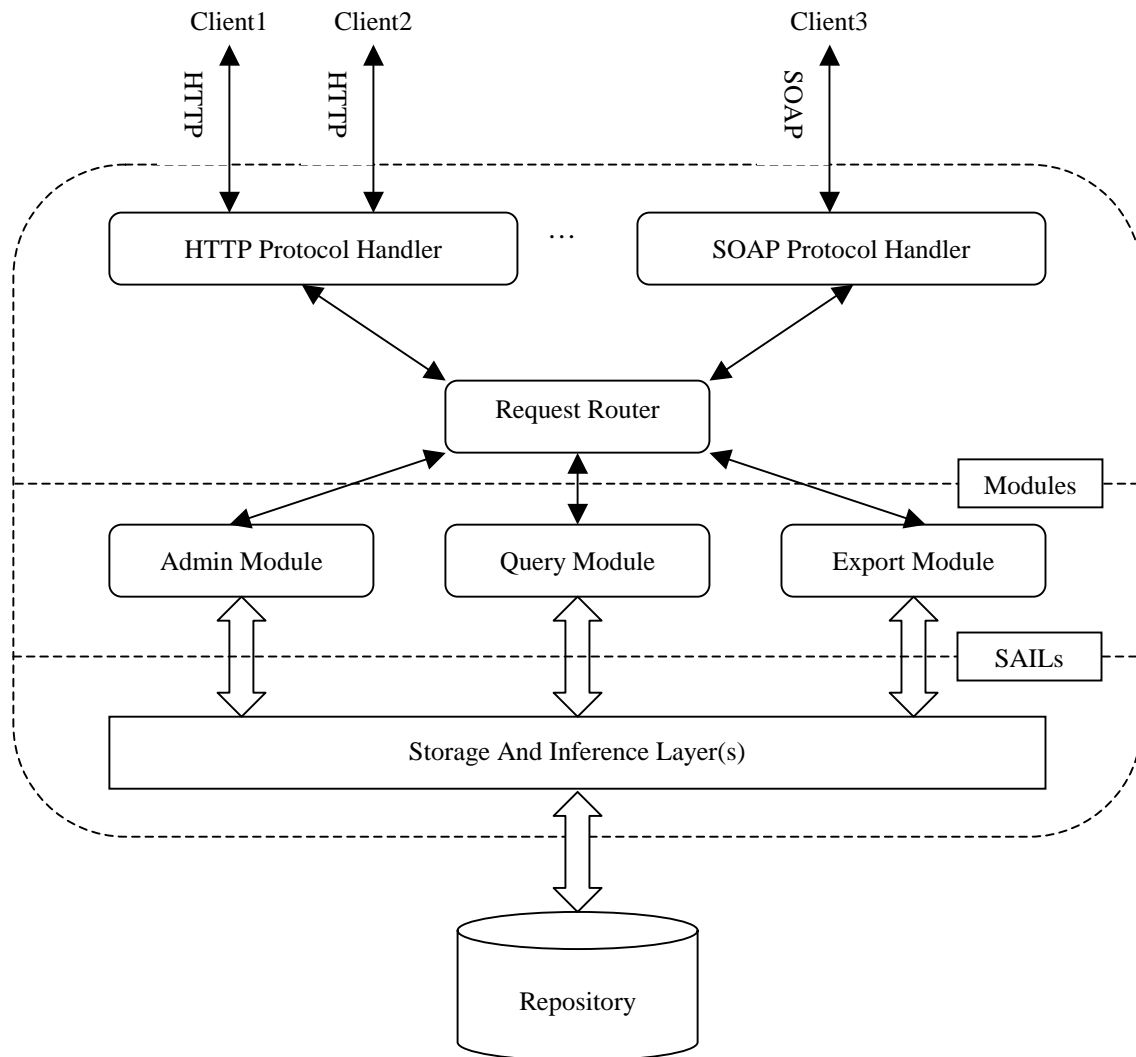
Another important change in the initially proposed design is that there will be a DAML+OIL reasoner implemented from scratch as a part of the SESAME architecture. The motivation for this change can be found in sub-section 4.1.3.

5.2 Overview of the Current SESAME Architecture

The current SESAME architecture is composed of several layers. The access layer is responsible to provide all the necessary functionality through the supported access protocols. 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. Follows a layer consisting of various modules each of which implements the basic functionality via calls to storage and inference layer (SAIL).

The SAIL interface hides the specific implementation dependant to the underlying physical storage. A number of SAILs can be stacked on top of each other, each of them handling some of calls and passing the others to the next 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].



5.3 How OMM Fits in the Picture?

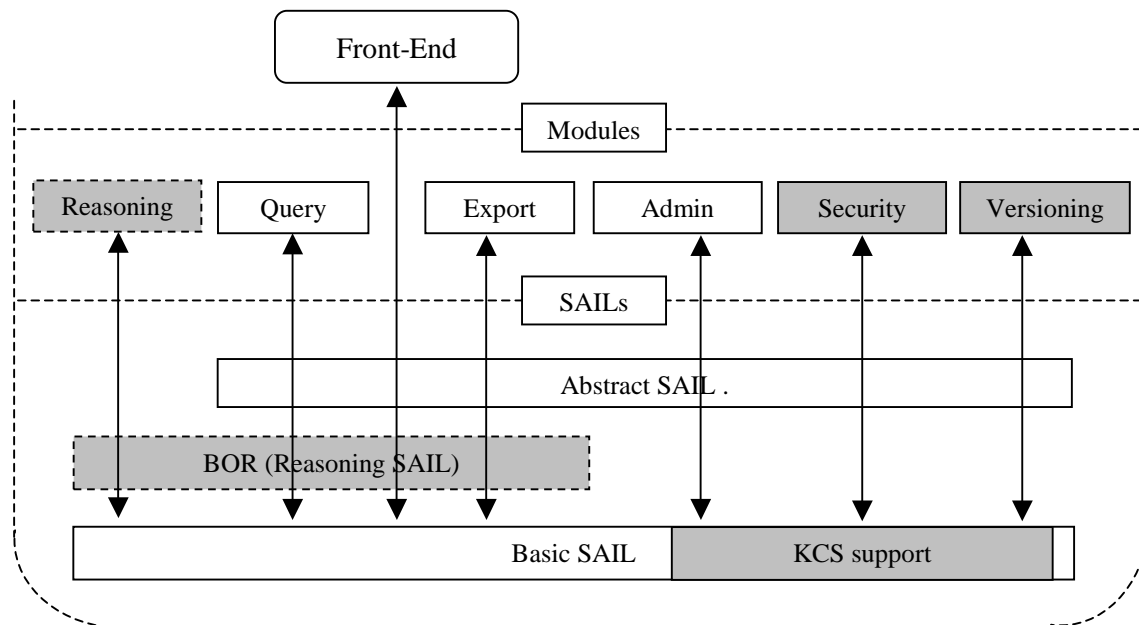
The interfaces of the OMM are designed to fit in the stacked SAIL architecture by extending it with their specific functionality. Some of the SESAME functional modules should be modified so to benefit directly from the interfaces in the OMM SAIL extension. The implementation of BOR (the DAML+OIL reasoner) will be added as an additional SAIL. Its full potential will be accessible via a separate Reasoning module responsible for routing the appropriate calls to the reasoner. On the other hand the reasoner will support the SAIL interface so the rest of the modules can also interoperate with it like with the standard RDF(S) supporting SAIL. For instance, the Query module will be 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) will have to be modified so to take care at the lowest level for some of the kinds of meta-information, especially those necessary for the knowledge control system (KCS). The reason for this design is that it is extremely important the information related to the access rights and “history” of the statements in the repository to be stored in the most efficient way that is why.

The proposed architecture will allow a transparent manipulation of the repository for the existing tools. Each application will work with the repository either by the existing modules (as it currently does) or by changing the repository behavior working directly with the new reasoning, versioning,

and security modules or gaining the access to the underlying low-level programming interfaces.

The scheme below represents how the lower part of the current SESAME architecture will be changed. The Request Router will also undertake modifications so to be able to (i) route the new interfaces and (ii) support more protocols. There are also important changes to take place in the router in order to enable the most convenient access control mechanism for each of the specific protocols (as explained further in sub-section 5.6.)



The gray boxes denote the new modules or extensions to be developed. The arrows depict how the requests of the different modules are passing through the different SAILs. The boxes with dashed borders (Reasoning and BOR) are optional, i.e. not an obligatory part of the SESAME architecture. Those are only necessary for DAML+OIL reasoning.

Another new feature is that front-end applications (such as editors and viewers) will be 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) will be handled through the request router. It also means that such requests will be subject of the standard access control and be possible through all the supported protocols.

5.4 Meta-Information

As discussed in sub-section 2.4, meta-information is supported for statements and resources. It is formally represented as RDF(S) itself so it can be queried via RQL. In addition the SAIL interfaces will be extended so to support meta-information to be directly retrieved for statements and resources. For this purpose the StatementIterator and ResourceIterator interfaces will be extended with the following method:

- `getMetainfo()` – returns a map of key-value pairs of types Resource and Value.

Special methods are not necessary for adding or removing meta-information, as far, as it can be manipulated as regular statements. For instance, a `status` meta-property for a resource `R` has to be set, it can be done in terms of adding the `<R, status, some_value>` statement. Obviously, if necessary the meta-information could be removed or changed in the same fashion.

5.5 Versioning Components, Version Management, and Tracking Changes

The interfaces and behavior described here are based on the versioning model for a Knowledge Control System (KCS) specified in section 2. Let us first discuss the most basic feature of a system with versioning – to provide access to different versions. In order to allow access to all possible versions or internal states of a repository each read operation should identify the desired state of the repository to which it must be applied. By default the read operation assumes the current state of the repository, i.e. it behaves as a system without any versioning features, so, the compatibility with tools currently using SESAME is assured. This default behavior also makes the versioning transparent for applications where it is not relevant.

A simple way for an application to take benefit from the versioning is to force the repository to provide view to an old state which to be used for the subsequent read operations. The exact state to be “seen” can be specified either by its update counter (UID) or by version, which is basically a labeled state of the repository. As explained in sub-section 2.5, the update counter represents the state of the repository when a particular addition or removal of a statement occurred, so, the set of the valid statements for a read operation for a specific state is reduced to those that were “alive” or present at this state, i.e. for which the state’s UID is between the UID’s recorded in their **omm:bornAt** and **omm:diedAt** meta-properties.

The functional interface to the versioning module can be logically separated into two groups of services to be provided which are closely related to the main tasks it is designed to solve – management of the distinct versions in the repository on one hand and a set of services allowing to track the changes between two states of the repository or access to the meta-information associated with a particular value of the update counter on the other.

The management of the meta-information for states and versions and the functionality to lock and unlock statements in the repository is part of the versioning module. The access to the interfaces `VersioningManagement` and `TrackingChanges` is made via the main KCS interface (see KCS) because that access is regulated at user level and the user should have the required permissions for these operations. The KCS interface is explained in the next section, as it is a more natural to present it with the security interfaces.

5.5.1 *VersionManagement interface*

- **labelState (UID)** – to create a labeled state of the repository (version) assigning the related meta-information for the new version
- **labelCurrentState ()** – to create a labeled state of the current version of the repository
- **revertToState (UID)** – to reset the repository to a previous state discarding all the changes made to it after that particular moment
- **workWithState (UID)** – all subsequent read operations to the repository to be made as it was in the given state
- **getVersions ()** – to retrieve all labeled states of the repository
- **lockStatements (statementsList)** – perform locking of statements in the repository
- **unlockStatements (statementsList)** – perform unlocking of statements in the repository

The value of the update counter and the associated meta-information about particular labeled state of the repository is accessed via the `Version` interface.

5.5.2 *Version interface*

- **getState ()** – retrieve the value of the update counter when the state is labeled

- **getMetaInformation()** – retrieve the meta-information associated with the version. The result is a Map with (property, value) pairs

5.5.3 TrackingChanges interface

- **getStatementHistory(statement)** – returns the tracking information about a statement in the repository
- **getRepositoryHistory()** – retrieve the meta-information collected during the life-time of the repository
- **pauseCounterIncrement()** – low level service to ‘stop’ the update counter so the next changes of the repository to be tracked together
- **continueCounterIncrement()** – low level service to continue with the ‘normal’ tracking of the changes in the repository one by one
- **isPausedCounterIncrement()** – to query the state of the increment behaviour of the update-counter
- **retrieveMetaInfo(state)** – retrieve the meta-information about particular state of the repository
- **clearHistory(UID, method)** – clear the dead statements, versions and related meta-information from the repository. There are two methods of clearing the history. The first one is to create a new Calendar - this means that the values of the update counter will be rearranged and all statements/resources, which are born before that given moment, are associated with ‘the moment just before that’. That moment is internal and all meta-information about the statements and resources is associated to it. The second one is to leave the calendar as it is without any additional rearrangement of these values.
- **aggregateUpdates(UID1, UID2)** - aggregate a number of sequential updates (say, between UID1 and UID2) to be merged and made equivalent to specified one, say UID3. In this case, all references to UIDs between UID1 and UID2 will be replaced with UID3, which may or may not be equal to UID1 or UID2.

5.6 Security Components. Architecture

Each operation to the repository is made on behalf of a user. Thus for each operation the user should be known. Each user has id and password that are used for authentication of the requested operation or creation of a session. For details on the security model, see section 3.

The user authentication is made via the Security manager and as result the roles assigned to the user are revealed with all the permissions associated with them. For the access protocols that do not support a ‘session’ this authentication should be made on each operation to the repository.

The permissions associated with the assigned roles to the user are used internally to manage the set of valid operations to the repository or to filter the data constructing a restricted view over the contents held in the repository.

The administration of the security information for the repository is made via the services provided by the security manager – one of the new modules added to the SESAME architecture. This includes services for:

- Addition or removal a users;
- Management of the roles assigned to a user;
- Addition and removal of roles and security classes;
- Create and assignment of security rules to roles, management of the role hierarchy.

The next sections present each set of services in the security manager organized in the following topics: user authentication, users management, roles management, and security rules. Lock and unlock of statements is considered as part of the version management and addressed in sub-section 5.5.

5.6.1 KCS Interface

The main entry point for all services of the Ontology Middleware Module is the interface KCS. Via that interface the user is authenticated to the system and also he can gain access to the VersionManagement, SecurityServices, TrackingChanges and Reasoning interfaces. Here follows a list of methods in the KCS interface:

- **authenticate(userid, password)** – checks the provided user information to locate the roles assigned to the user. Afterwards the security rules for these roles are used to process the requested operation on behalf of the user.
- **getSecurityServices()** – access the SecurityServices interface allowing manipulations of the users and roles defined in the repository.
- **getTrackingChanges()** – access the TrackingChanges interface allowing manipulation of the history of the repository
- **getVersionManagement()** – access the VersionManagement interface to manipulate distinct versions and states of the repository. Also the user can lock or unlock parts of the repository from there.
- **getReasoner()** – access the Reasoner interface

5.6.2 SecurityServices Interface

The SecurityServices interface provides methods to organize the access to the repository managing the users and roles defined in it. The purpose of the following methods is to manage the users that have access to the repository. They are separated semantically to these two distinct groups:

5.6.2.1 Users Management

- **addUser(userid, password)** – create a new user
- **removeUser(userid)** – remove a user
- **getUsers()** – retrieve a list of all users
- **getUser(userid)** – access the interface allowing manipulations about particular user

5.6.2.2 Roles Management

- **createRole(roleid, parentRoles)** – create a new role
- **removeRole(roleid)** – remove existing role.
- **getRoles()** – retrieve a list of all roles
- **getRole(roleid)** – access the interface allowing manipulations about particular role

5.6.2.3 Security Class Management

- **createSecurityClass(scid, label)** – create a new security class
- **removeSecurityClass(scid)** – remove existing security class
- **getSecurityClasses()** – retrieve a list of all security classes for the repository

- **getSecurityClass (scid)** – access the interface allowing manipulations about particular security class

5.6.3 User Interface

The definition of each user in the repository can be accessed via the interface User. That interface has methods from which to manage the userID, password and assigned roles.

- **getID ()** – retrieve the userID of the user
- **setID (newID)** – change the userID of the user
- **getPassword ()** – retrieve the user password
- **setPassword (newPassword)** – change the user password
- **getRoles ()** – retrieve the list of roles assigned to the user
- **setRoles (rolesList)** – assign new roles to a user

5.6.4 Role Interface

- **getRoleID ()** – retrieve the id of the Role
- **setRoleID (newRoleID)** – change the id of the Role
- **addRule (rule)** – add a new rule to the role
- **removeRule ()** – remove a rule
- **getRules ()** – retrieve a list of rules associated with the role
- **getAllRules ()** – retrieve a list of all rules (including those inherited from the parent roles)
- **setParentRoles (rolesList)** – inherit Roles
- **getParentRoles ()** – retrieve the list of the parent Roles

5.6.5 Rules

The smallest piece of information used to control the access to the information in the repository is the security rule. Each instance of such a rule has associated set of access rights, which are applied to the set of objects in the repository. The restrictions can be defined to specific objects within the repository as it is explained in SPR4 in section 3.4. The basic operations allowed to a rule are to add, remove and retrieve set of objects using one of the provided restriction types and to access and change the set of access rights associated with it.

- **addRight (Right)** – assign an access right to the rule;
- **removeRight (Right)** – remove an access right from the rule;
- **getRights ()** – retrieve a list of rights granted by the rule;
- **setRestriction (Restriction)** – add restriction to set of objects in the repository;
- **getRestriction ()** – retrieve the restriction of the rule.

5.6.6 Restriction Interfaces

For each of the restriction types mentioned in SPR4 there exist a separate interface, which is used to define and explore it. All this interfaces inherit the Restriction interface that.

- **Restriction** – this is a base interface that only provides information about the type of the restriction.

- `getType()` – add a class to the restriction
- **RepositoryRestriction** – this interface do not have any additional methods since its role is to indicate that the access rights in a rule are applicable for all objects in the repository.
- **SchemaRestriction** – this interface also do not introduce any new methods. Its purpose is to indicate that the access rights are applicable to all resources or statements recognized as part of the schema
- **ClassesRestriction** – the interface define all the resources (instances) of specific classes, including the statements where those are subjects. Resources of sub-classes also considered. The basic operations are addition, removal and access to the class names.
 - `addClass(classId)` – add a class to the restriction
 - `removeClass(classId)` – remove a class from the restriction
 - `getClasses()` – retrieve the list of classes in the restriction
- **InstancesRestriction** – interface to define a set of resources, including the statements where those are subjects
 - `addResource(resourceId)` – add a resource to the restriction
 - `removeResource(resourceId)` – remove a resource from the restriction
 - `getResources()` – retrieve the list of resources in the restriction
- **PropertiesRestriction** – interface to define a set of the statements with specific properties as predicates. Sub-properties are also included.
 - `addProperty(propertyId)` – add a property to the restriction
 - `removeProperty(propertyId)` – remove a property from the restriction
 - `getProperties()` – retrieve the list of properties in the restriction
- **PatternRestriction** – interface to define a composite set of statements where they will be matched according to the subject and object via InstancesRestriction or ClassesRestriction and to their predicate via a PropertyRestriction. This interface allow manipulation of the subject, predicate and object restrictions
 - `setSubjectRestriction(resourceRestriction)` – set a ResourceRestriction to the subject of the pattern restriction. If the parameter is `null`, the subject restriction is cleared.
 - `setPredicateRestriction(propertyRestriction)` – set a PropertyRestriction to the predicate of the pattern restriction. If the parameter is `null`, the predicate restriction is cleared.
 - `setObjectRestriction(resourceRestriction)` – add a restriction to the object part of the pattern restriction. If the parameter is `null`, the object restriction is cleared.
 - `getSubjectRestrictions()` – retrieve a list of restriction applied to the subject of the pattern restriction
 - `getPredicateRestrictions()` – retrieve a list of property restrictions applied to the predicate of the pattern restriction
 - `getObjectRestrictions()` – retrieve a list of restrictions applied to the object part of the pattern restriction
- **QueryRestriction** – interface to specify a set of statements or resources as result of a RQL query.

- `setQuery(queryString)` – set the defining query of the restriction
- `getQuery()` – get the defining query of the restriction

5.6.7 SecurityClass Interface

- `getID()` – retrieve the id of the security class
- `setID(newID)` – change the id of the security class
- `setQuery(queryString)` – set the defining query of the class
- `getQuery()` – get the defining query of the class
- `setUpdatePeriod(duration)` – set the duration between two “updates” of the security class, see section 3.5.2. The duration specified in seconds.
- `getUpdatePeriod()` – get the update period of the class. The duration is returned in seconds.
- `forceUpdate()` – force an “update” of the security class, see section 3.5.2.

5.7 Reasoner interface

The methods in the Reasoner interface are discussed in sub-section 4.7. The DAML API provided in the JENA toolkit (<http://www.hpl.hp.com/semweb/daml.html>) will be used to represent the DAML+OIL language constructs where they appear as parameters for results.

6 References

- [**Bechhofer et al. 1999**] Sean Bechhofer, Ian Horrocks, Peter F. Patel-Schneider, Sergio Tessaris.
A Proposal for a Description Logic Interface.
In: (edited by: Patrick Lambrix, Alex Borgida, Maurizio Lenzerini, Ralf Mueller, Peter Patel-Schneider) Proceedings of the Intl. Workshop DL'99, Linköping, Sweden, July 30 - August 1, 1999.
- [**Benatallah and Tari, 1998**] Boualem Benatallah, Zahir Tari
Dealing with Version Pertinence to Design an Efficient Schema Evolution Framework.
In: Proceedings of a "International Database Engineering and Application Symposium (IDEAS'98)", pp.24-33, Cardiff, Wales, U.K. July 8-10 1998
- [**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/download/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/download/del10.pdf>
- [**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/download/del4.pdf>
- [**Das et al, 2001**] Aseem Das, Wei Wu, Deborah L. McGuinness, and Adam Cheyer
Industrial Strength Ontology Management for E-Business Applications.
In the Proc. of International Semantic Web Working Symposium (SWWS), July 30 - August 1, 2001, Stanford University, California, USA.
- [**Dimitrov, 2000**] Dimitrov, Marin
XML Standards for Ontology Exchange.
In: Proceedings of "OntoLex 2000: Ontologies and Lexical Knowledge Bases", Sozopol, Sept. 8-10, 2000.
- [**Franconi et al, 2000a**] Enrico Franconi, Fabio Grandi, Federica Mandreoli
Schema Evolution and Versioning: a Logical and Computational Characterization
In "Database schema evolution and meta-modeling" - Ninth International Workshop on Foundations of Models and Languages for Data and Objects, Schloss Dagstuhl, Germany, September 18-21, 2000. LNCS No. 2065, pp 85-99
- [**Franconi et al, 2000b**] Enrico Franconi, Fabio Grandi, Federica Mandreoli
A Semantic Approach for Schema Evolution and Versioning of OODB
published in the Proceedings of the 2000 International Workshop on Description Logics (DL2000), Aachen, Germany, August 17 - August 19, 2000. pp 99-112
- [**Haarslev et al, 2001**] Haarslev V., Mueller R., Turhan A.-Y.
Exploiting Pseudo Models for TBox and ABox Reasoning in Expressive Description Logics. IJCAI2001
- [**Hayes, 2001**] Patrick Hayes
RDF Model Theory.

W3C Working Draft. <http://www.w3.org/TR/rdf-mt/>

- [**Horrocks and Sattler, 2001**] Horrocks I. and Sattler U.
Ontology Reasoning in the SHOQ(D) Description Logic, IJCAI2001
- [**Jajodia, 2001**] Sushil Jajodia, Pierangela Samarati, Maria Luisa Sapino, V. S. Subrahmanian
Flexible support for multiple access control policies.
ACM Transactions on Database Systems (TODS) , Volume 26, June 2001, pp.214-260.
- [**Jones, 1996**] - V. Jones,
"Access Control for Client-Server Object Databases", Ph.D. Thesis, Department of
Computer Science, University of Illinois, December 1996.
- [**Kiryakov et al, 2001**] Atanas Kiryakov, Kiril Iv. Simov, Marin Dimitrov.
OntoMap - the Guide to the Upper-Level.
In: Proceedings of the International Semantic Web Working Symposium (SWWS), July
30 - August 1, 2001, Stanford University, California, USA.
- [**Kitcharoensakkul and Wuwongse, 2001**] Supanat Kitcharoensakkul and Vilas Wuwongse
Towards a Unified Version Model using the Resource Description Framework (RDF)
International Journal of Software Engineering and Knowledge Engineering (IJSEKE),
Vol. 11, No. 6 (December 2001)
- [**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/>
- [**ORACLE, 2001**] – ORACLE Corporation
"Oracle Label Security Administrator's Guide, Release 9.0.1" Part Number A90149-01
http://download-west.oracle.com/otndoc/oracle9i/901_doc/network.901/a90149/title.htm
- [**Pan and Horrocks, 2001**] Pan, Jeff; Horrocks, Ian
Metamodeling Architecture of Web Ontology Languages.
In the Proc. of International Semantic Web Working Symposium (SWWS), July 30 -
August 1, 2001, Stanford University, California, USA.
- [**Patel-Schneider, 2001**] Peter Patel-Schneider, editor
A Model-Theoretic Semantics for DAML+OIL (March 2001).
<http://www.daml.org/2000/12/model-theoretic-semantics.html>, November 2001.
- [**Patel-Schneider and van Harmelen, 2001**] Patel-Schneider P., and van Harmelen Fr.
Coordination points between RDF(S) and DAML+OIL.
<http://www.daml.org/2001/07/RDFS-DAML+OIL-coordination.html>
- [**Simov, 2002**] Kiril Iv. Simov
DAML+OIL Syntax Overview and Analysis.
Technical Report, OntoText Lab, February 2002.
<http://www.ontotext.com/publications/daml-oil-syntax.htm>
- [**Schneider, 1998**] F. Schneider
Enforceable Security Policies.
TR 98-1664, Department of Computer Science, Cornell University, Ithaca, NY, 1998.
- [**van Harmelen et al. 2001**] Frank van Harmelen, Peter F. Patel-Schneider and Ian Horrocks,
editors
Reference description of the DAML+OIL (March 2001) ontology markup language.
<http://www.daml.org/2001/03/reference.html>
- [**Wijesekera and Jajodia, 2001**] D. Wijesekera, S. Jajodia
Policy Algebras for Access Control - The Propositional Case
Proc. of the ACM Conference on Computer and Communications Security, November 5-
8, 2001.