Ontology Middleware: Platform for Real-World Knowledge Management

Atanas Kiryakov[1], Damyan Ognyanov[1], Kiril Simov[2,1], Borislav Popov[1], Stanislav Jordanov[1]

[1] OntoText Lab, Sirma AI EOOD, 38A Chr. Botev blvd, 1000 Sofia, Bulgaria
{naso, damyan, borislav, stenly}@sirma.bg
[2] Linguistic Modelling Lab, CICT, Bulgarian Academy of Sciences, 25 Acad. G. Bontchev str, 1113 Sofia, Bulgaria
kivs@bgcict.acad.bg

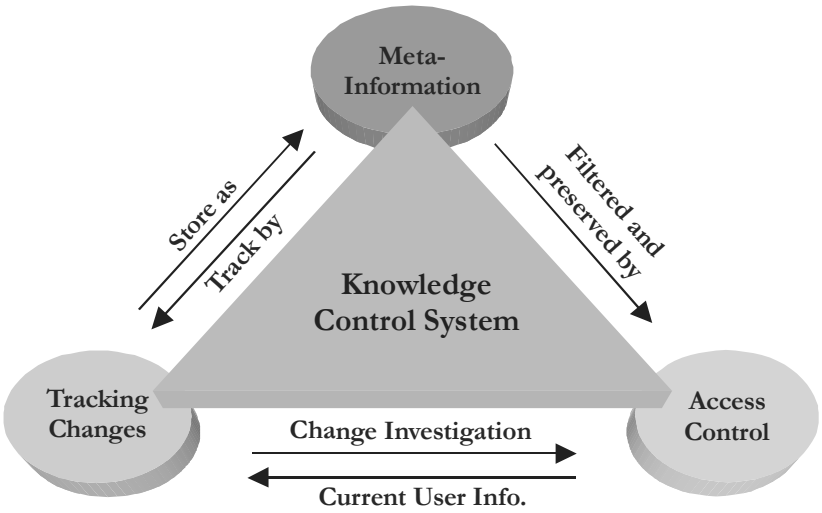
Abstract. An ontology middleware system is defined, presented, and discussed in this paper. It includes number of features critical for the implementation of real-world knowledge management application: versioning, fine-grained access control, and meta-information. Important part of the research presented is the definition of proper models for support for each of these aspects. A security model with three levels of complexity is designed so to support proper management of the compromise between complexity and efficiency. The system presented is under development as a part of the On-To-Knowledge project where it is implemented as extension of the Sesame repository and allows management of knowledge represented in RDF(S)-based languages, including DAML+OIL. Although the implementation makes strong commitments to RDF(S), the definitions of the change tracking and security problems are of general KR importance. This paper is further development of the results reported in [11].

1. Introduction

The ontology middleware presented here can be described as an „administrative“ software infrastructure that makes the rest of the modules in a knowledge management toolset 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 large knowledge bases. The following features are considered:

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

These three aspects are interrelated as depicted on the following scheme.



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.

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. An example of such a system is the Sesame RDF(S) repository, [4];
- Multi-protocol client access to allow different users and applications to use the system via the most efficient "transportation" media;
- Knowledge control – the KCS introduced above;
- Support for plugable reasoning modules suitable for various domains and applications. This ensures that one and the same system may be used within single enterprise or computing environment for various purposes (that require different reasoning services.)

The work presented here was carried as part of the On-To-Knowledge project. The design and implementation of the ontology middleware module is an extension of the Sesame architecture (see [4]) that already covers many of the desired features. Earlier stage of the research is presented in bigger details in [11]. The later report also presents a DAML+OIL reasoner, called BOR, developed as a plugable module for Sesame/OMM.

The ontology middleware module presented extends the Sesame RDF(S) repository that affects the management of both ontologies and instance data in a pretty

much unified fashion. Here the term **repository** will also be used to denote a compact body of knowledge that could be used, manipulated, and referred as a whole.

Section 2 is dedicated to tracking changes in RDF(S) repositories – definition of the task, related work, principles, design and implementation approach. In a similar fashion section 3 targets the security issue. Future work and conclusion follow in the last section.

2. Tracking Changes, Versioning, and Meta-Information

In this section, we address the problem for tracking changes within a knowledge base. Higher-level evaluation or classification of the updates (considering, for instance, different sorts of compatibility between two states or between ontology and old instance data) is beyond our scope – those are studied in depth in [5]. The tracking of the changes in the knowledge (as discussed here) only provides the necessary basis for further analysis. 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 comment several studies related to versioning of a complex data objects. Although some of the sources discuss similar problems 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 capturing of the interactive changes in knowledge repositories such as assertions and retractions.

Database schema evolution and the tasks related to keeping schema and data consistent to each other can be recognized as a very similar problem. A detailed and pretty formal study on this problem can be found in [7, 8] – it presents an approach allowing the different sorts of modifications of the schema to be expressed within suitable description logic. Another study dealing with the design of a framework handling database schema versioning is presented in [1] – it presents a different approach of handling the changes of the evolving objects and classes.

Probably the most relevant work was done under the On-To-Knowledge project – among the reports concerning various aspects of the knowledge management, most relevant is [5], mentioned earlier in this section.

2.2. Versioning Model for RDF(S) Repositories

A model for tracking of changes, versioning, and meta-information for RDF(S) repositories is proposed. The most important principles are presented in the next paragraphs.

VPR1: *The RDF statement is the smallest directly manageable piece of knowledge.*

Each repository, is a set of RDF statements (triples) – the smallest separately manageable pieces of knowledge. Arguments exist that the resources and the literals are the smallest entities – it is true, however they cannot be manipulated independently, only as part of a statement. 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.*
Since statements are nothing more than triples, changing one of the constituents, just creates another triple by removing the intitial one and adding the resulting one.

VPR3: *The two basic types of updates are addition and removal of a statement*
Those are the events that have to be tracked by a tracking system. Event types such as replacement or simultaneous addition of statements can all be seen as composite events that can be modeled via sequences of additions and removals.

VPR4: *Each update turns the repository into a new state*
A state of the repository is determined by the set of explicitly asserted statements. 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.

2.2.1. 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. Obviously, 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. We assumed that there could be equivalent states in the history of a repository, but they are still managed as distinct entities (see [11] for motivation.)

2.2.2. 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. Once defined to be a version, the state becomes a first class entity for which additional knowledge could be supported as a meta-information.

2.3. Meta-Information

Meta-information here is discussed as a kind of information that does not change the semantics of the "real" knowledge. Such could be information about the status of development, comments and documentation, maintenance hints, etc. The information related to tracking of changes is also a meta-information. In the context of the ontology middleware presented here, meta-information is supported for resources, 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. More detailed discussion about the support of meta-information can be found in [11], in this sub-section we are just mentioning the central issues:

- The meta-information can be seen as RDF(S) itself. This means that it can be queried and extracted in arbitrary combination with the "real" information. It is however the case that internally, some of the meta-information is stored in a different way for performance and security reasons. Thus the system supports a kind of mimicry to make it available as RDF(S);
- All the KCS related information would be represented in RDF according to http://www.ontotext.com/otk/2002/03/kcs.rdfs. That includes tracking, versioning, and security information as well as user-defined meta-information. 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`**. This approach allows for easy filtering and separation of the meta-information, although it can easily be used together with the "real" one;
- Changes in the meta-information are considered as changes of the state of the repository and hence they are tracked. The only exception is the tracking meta-information itself[1];
- There is a special reification-like mechanism developed to support user-defined meta-information about statements. As far as reification is not formally used there are no problematic requirements for the repository. The mechanism is tested to work smoothly with Sesame, details could be found in [11][2].

2.4. Implementation Approach

The schema for tracking changes in a repository is described first. For each repository, there is an *update counter* (UC) – an integer variable that increases its value each time the repository is updated. In the basic case, that means when a statement gets added or deleted. Let us call each separate value of the UC *update identifier, UID*. The UIDs of thew adding and removal are known for each statement – these values determine the "lifetime" of the statement. It is also the case that each state of the repository is identified by the corresponding UID.

The UIDs that determine the "lifetime" of each statement are kept, so, for each state it is 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 unique UID.

The approach could be demonstrated with the sample repository KB1 and its "history". The repository is represented as a graph, the lifetime of the statements is given after the property names. The history is presented as a sequence of events in format :

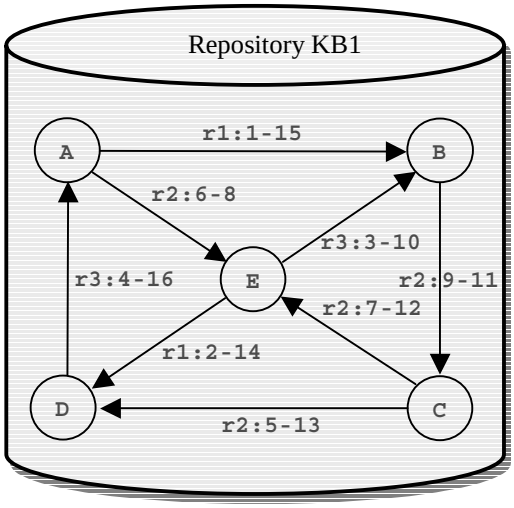
[1] This allows paradoxes like the one with the Russell's barber to be avoided.

[2] A full working example can be found at http://www.ontotext.com/otk/statement_metainfo_ex.rdf

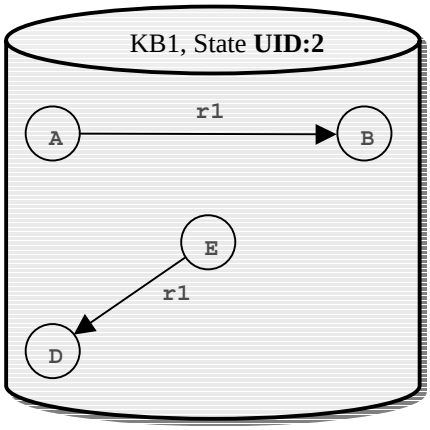
```
UID:nn {add|remove} <subj, pred, obj>
```

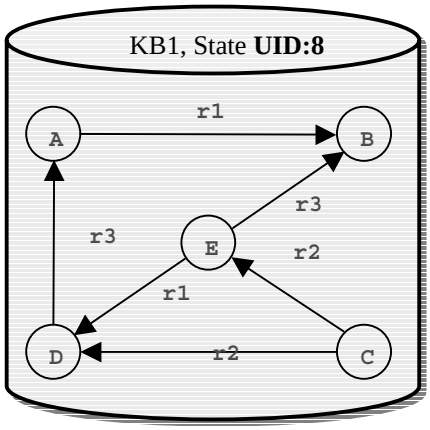
History:

```
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 to handle in the above model, multiple additions and removals of one and the same statement, which in a sense periodically appears and disappears form 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 states.

2.4.1. Batch Updates

We call batch update the possibility for the update counter of the repository to be stopped, so not to increment its value for a number of consecutive 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. Another reasonable example for a batch update is 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.

2.4.2. 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 are treated as read-only. All these statements are added and removed to/from the repository simultaneously with the statement that causes their inference or import.

2.4.3. 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 **`ST1`** in the repository, there was another statement added, namely **`ST4=<B, rdfs:subClassOf, D>`**. As far, as **`ST2`** is already in the repository only **`ST5=<D, rdf:type, rdfs:Class>`** will be inferred and added. Now, imagine **`ST1`** is deleted next while **`ST4`** remains untouched. Should we delete **`ST2`**? It was added together with **`ST1`** on one hand, but on the other it also follows from **`ST4`**. Resolving such problems requires the so-called "truth maintenance systems" (TMS) – basically, for each statement (and most important, for the inferred ones) information is being kept about the statements that "support" it, i.e. such that (directly) lead to its inference. A TMS is currently in development for Sesame – its design is an interesting research problem related to the model-theoretic semantics of RDF(S). An important characteristic of RDF(S) is that the each time when a statement is inferred it is possible to determine a group of up to tree statements that caused the inference, i.e. jointly support the inferred statement. The TMS it is not presented here dew to space limitations.

Assuming, there is a TMS working, 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 group of 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.

2.4.4. Versioning of Knowledge Represented in Files

Here we only consider the import of knowledge into the Sesame from files. 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. Three different modes for populating repository 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 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 may lead to inconsistency of the repository even if both the previous state and the file was consistent;
- **Updating** – after the import the repository contains only the statements form the file, the set **`FS`** (as in the re-initializing mode). The difference is that the statements from the repository that were not in **`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 **`FS`** that were not already in the repository[3] 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.

2.4.5. Branching Repositories

Branching of state of repository is possible – technically a new repository is created and populated with a certain state of the existing one, we want to make branch of. When a state is getting branched, it is automatically 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.

[3] Actually those that are not "alive".

3. Security and Access Control

Here we define a model for access control over RDF(S) repositories that allows development of a system handling ontologies, instance data, and knowledge bases of various types in the most unrestrictive way. The requirements for such security system are discussed in detail in [11] together with proper argumentation about the importance of the support of fine-grained security rules for encoding of business logic. The formal representation of the security data together with the implementation approach is discussed in the last sub-section.

3.1. Related Work

There is almost no related work considering directly access control systems for knowledge bases, ontology management systems, or repositories. In [6] the Ontology Builder and Ontology Server products are presented, with short discussion on the security system included – the access rights there are defined as 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. The standard security schemata there provide control of the access down to the level of a database instance and distinct tables. In contrast, our security system to provides means for controlling the access with much better granularity, namely at the level of instances or records. An interesting exception presenting access control down to record level is the Oracle Label Security presented in [13]. A general discipline called Role-based Security covers wide range of theoretical issues related to the design of various access control systems – our research is follows the lines of [9].

3.2. Basic Principles of the Security Model

The basic principles underlying a security model for RDF(S) are presented here. Those in a high degree determine also the implementation approach and the representation of the security information discussed in the next sub-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.

***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 3.2.1. . Discussion on the different restriction types can be found in sub-section 3.2.3. .

***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 a user 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 repository[4] – 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. There are also no separately defined rights for import and export – those are consequence of the *Add* and *Read* rights.

The *History* right allows management of the tracking information; this includes labeling versions and the kind of modifications discussed in sub-section 2.2.1. *Admin* and *History* rights are supported just on the repository level.

[4] Thanks to Michel Klein, who raised this issue within a discussion.

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 related security rules to be grouped together. 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.

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.*

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 [9]) in the terms of the security model introduced here.

3.2.1. Security Classes

Security classes are defined via RQL queries. Those allow specification of sets of resources[5] that correspond to complex criteria more expressive than Patterns.

The deviation of the security class notion from the class notion in RDF(S) follow:

- 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 considered an instance of a security class if it is member of the result of its defining query.

3.2.2. Query Restrictions

The Query restrictions can bear unrestricted complexity including context-related logic that depends on the user making the request.

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.

3.2.3. 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;
- *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;
- *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.

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

[5] 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.

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.

3.3. Implementation Approach

The implementation approach for the different aspects of the security enforcement is discussed in [11]. Finally, the schema for the formal representation of the security information is presented.

3.3.1. Efficiency Assumptions

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.

Under the above assumptions there exists efficient procedure (described in [11]) that also supports streaming operations and requires two significant 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. Details about the implementation approach for support of Security Classes and Query restrictions are provided in [11].

3.3.2. 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 might be kept natively in a different format due to efficiency or security reasons. The important point is that such data logically follow this schema and can be extracted in RDF(S) if necessary.

4. Conclusion and Future Work

The ontology middleware module presented still have to prove itself in real-world applications. At this stage it is work in progress inspired by the methodology, tools, and case studies developed under the On-To-Knowledge project. The reasoner of the OntoMap (see, http://www.ontomap.org/) project will be wrapped as a SESAME storage and inference layer – it will provide an interesting alternative between RDF(S) and DAML+OIL, as well, as another justification for the capabilities of the proposed design to "host" various inference services under single hat.

5. References

1. 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
2. W3C; Dan Brickley, R.V. Guha, eds. *Resource Description Framework (RDF) Schemas.* http://www.w3.org/TR/2000/CR-rdf-schema-20000327/
3. Jeen Broekstra, Arjohn Kampman. *Query Language Definition.* Deliverable 9, On-To-Knowledge project, May 2001. http://www.ontoknowledge.org/downl/del9.pdf
4. Jeen Broekstra, Arjohn Kampman. *Sesame: A generic Architecture for Storing and Querying RDF and RDF Schema.*Deliverable 10, On-To-Knowledge project, October 2001.
   http://www.ontoknowledge.org/downl/del10.pdf
5. Ying Ding, Dieter Fensel, Michel Klein, Borys Omelayenko. *Ontology management: survey, requirements and directions.* Deliverable 4, On-To-Knowledge project, June 2001. http://www.ontoknowledge.org/downl/del4.pdf
6. 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.
7. 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
8. Enrico Franconi, Fabio Grandi, Federica Mandreoli. *A Semantic Approach for Schema Evolution and Versioning of OODB.* Proceedings of the 2000 International Workshop on Description Logics (DL2000), Aachen, Germany, August 17 - August 19, 2000. pp 99-112
9. 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.
10. Patrick Hayes. *RDF Model Theory*. W3C Working Draft. http://www.w3.org/TR/rdf-mt/
11. Atanas Kiryakov, Kiril Iv. Simov, Damyan Ognyanov. *Ontology Middleware: Analysis and Design.* Deliverable 38, On-To-Knowledge project, March 2002.
12. 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/
13. ORACLE Corp. *"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